BRAINDROP: A MIXED-SIGNAL NEUROMORPHIC ARCHITECTURE WITH A DYNAMICAL SYSTEMS-BASED PROGRAMMING MODEL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Alexander Neckar
June 2018

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kwabena Boahen, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Rajit Manohar,

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

Abstract

This thesis describes the architecture of *Braindrop*, a .85 mm^2, 4096-neuron, low-power, mixed-signal neuromorphic system. Braindrop is the first such system designed with a comprehensive set of high-level programming abstractions and a synthesis procedure for mapping them to mismatched (and temperature-sensitive) subthreshold analog hardware. This high level of abstraction stands in stark contrast to previous neuromorphic systems, which required expert knowledge (and extensive characterization) of the hardware to use. Braindrop's computations are specified as coupled nonlinear dynamical systems. This program specification is synthesized to the hardware using the Neural Engineering Framework, not just compensating for, but leveraging the fabric of mismatched analog circuit elements as dynamic computational primitives. For typical network configurations, Braindrop achieves an energy per equivalent synaptic operation of 388 fJ.

Contents

List of Tables

List of Figures

Chapter 1

Mixed-Signal Neuromorphic Systems

To improve the energy-efficiency of artificial neural networks, neuromorphic computing seeks to emulate the brain's harnessing of analog signals to efficiently compute and communicate. Biological neurons minimize energy by only sparingly emitting digital spikes to perform global communication. This sparseness is enabled by neurons' ability to continuously and dynamically update their analog membrane potentials. Analog signals also amortize digital communication's high cost by distributing spikes to dozens of targets at a time via local signal propagation in dendritic trees. This complementary relationship between digital and analog signaling does not exist in typical neural network implementations, where spike trains are replaced by binary numbers, neurons are turned into static nonlinearities, and synaptic connections are realized with matrix multiplication.

While analog circuitry promises energy efficiency because of its potential to sparsify global digital communication, when implemented in large-scale integrated circuits, its inherent variability (transistor mismatch and temperature sensitivity) impedes programmability and reproducibility. To achieve density and energy efficiency, silicon neurons must be sized as small as possible and use as little current as possible, leading to extreme mismatched behavior and temperature sensitivity. This variability is directly exposed to the user when mixed-signal neuromorphic systems are programmed on the level of neuron parameters and individual synaptic weights. This programming paradigm limits adoption to experts willing to understand the hardware at the circuit level. Furthermore, because each chip is different, for a given computation, the programming parameters are different for each one. In addition, their mismatch varies with temperature, compounding these challenges.

This thesis presents *Braindrop*, the first mixed-signal neuromorphic system designed with a set of mismatch- and temperature-invariant abstractions in mind. The user describes their computation as a system of nonlinear differential equations, agnostic to the underlying hardware. Synthesis proceeds by characterizing the hardware and implementing each equation using a group of neurons. This paradigm, inspired by the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2003), is not only tolerant of, but reliant on mismatch: neuron responses form sets of basis functions, which must be dissimilar. No individual neuron circuit or parameter is critically important. Thus, Braindrop's hardware and software not only leverage, but tame, the heterogeneity of its analog circuitry, presenting clean abstractions to the user.

1.1 The Neural Engineering Framework

The Neural Engineering Framework (NEF) is an appealing theoretical framework for neuromorphic hardware for a number of reasons. First, analog neuron heterogeneity is a boon rather than a hindrance. Second, analog synapse dynamics serve as a native computational primitive for implementing arbitrary dynamical systems. Third, a compressed version of the weight matrix—encoders and decoders—reduce on-chip memory requirements.

With the NEF, mapping computations to neurons proceeds as follows: first, the user decomposes their desired computation into a coupled system of subcomputations. Each subcomputation is implemented by a single group of neurons, a *pool*. The pool's activity encodes the input signal, which may be multidimensional. This encoding is accomplished by giving each neuron a preferred direction in the input space specified by an *encoding vector*. A neuron is excited (receives positive current) when the input points in the direction of the encoding vector, and is inhibited (receives negative current) when it points away. Given a varied selection of encoding vectors and a sufficiently large pool, the neurons' nonlinear responses form a basis set for approximating arbitrary multidimensional functions of the input space, simply by computing a weighted sum of the responses, a linear *decoding* (Figure 1.1 on page 3).

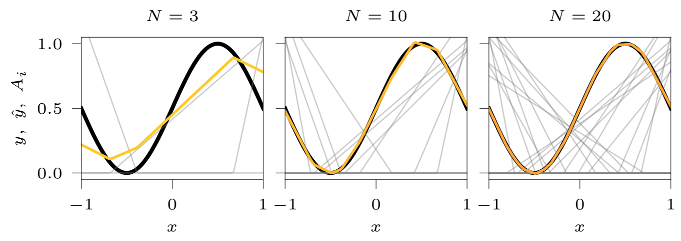


Figure 1.1: $y = [\sin(\pi x) + 1]/2$ (black) is approximated by $\hat{y} = Ad$ (yellow), where each column of A represents a single neuron's firing rates over the input range (gray) and d is obtained by solving for $\mathrm{argmin}_d ||Ad - y||_2 + \lambda ||d||_2 + \kappa ||d||_1$. $\hat{y}$ is plotted for decreasing values of κ, which allows the optimization to use more neurons (3, 10, and all 20), thereby decreasing the error. Thus, approximation error is a knob the user has at their disposal: more resources may be expended to achieve higher precision.

The multidimensional input is therefore projected by the encoder into a much higher-dimensional space, passed through the neurons' nonlinearities, and then projected by the decoder into another multidimensional space. Dynamic transformations, realized by recurrently connecting the output of the pool to its input, exploit the synapses' low-pass filtering operation. This approach allows arbitrary-order nonlinear dynamical systems to be implemented by a single pool (Eliasmith and Anderson, 2003). Pools are connected by linking the output of decoders to the inputs of encoders to form large network graphs (Figure 1.2 on page 4). Linear *transforms* may also be placed between decoders and encoders.

Computational errors arise from two sources: poor function approximation due to inadequate basis functions, and spurious spike coincidences (Poisson noise). Function approximation is generally improved when there are more neurons allotted to each pool, and is generally made more difficult as the dimensionality of the input space increases. Factoring large many-dimensional functions into several fewer-dimensional ones before mapping into pools can therefore be advantageous. For a fixed number of dimensions, function smoothness also relates to ease of approximation. Poisson noise is a function of the synaptic time-constant and the neurons' spike rates. A longer time-constant will produce a smoother filtered signal, but will introduce additional delay when cascading layers. These two error sources must be balanced against each other when factoring a high-dimensional function if there is a latency constraint: for a fixed total number of neurons, factoring may improve

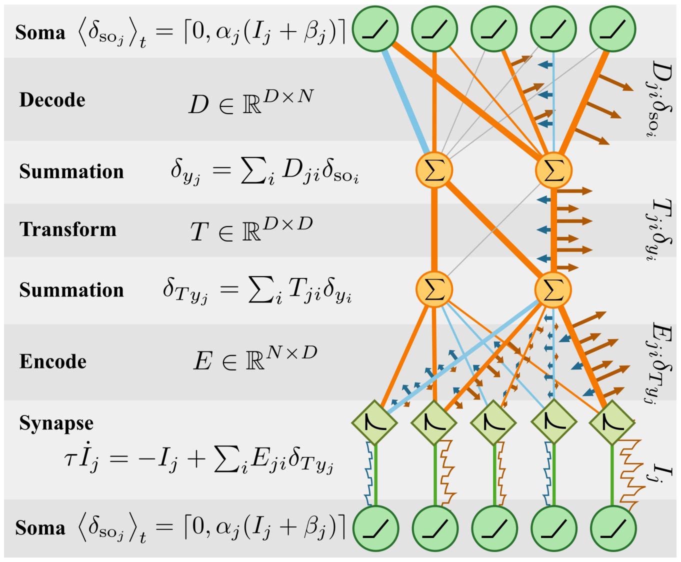

Soma $\langle \delta_{\text{so}_j} \rangle_t = \lceil 0, \alpha_j(I_j + \beta_j) \rceil$
Decode $D \in \mathbb{R}^{D \times N}$
Summation $\delta_{y_j} = \sum_i D_{ji} \delta_{\text{so}_i}$
Transform $T \in \mathbb{R}^{D \times D}$
Summation $\delta_{Ty_j} = \sum_i T_{ji} \delta_{y_i}$
Encode $E \in \mathbb{R}^{N \times D}$
Synapse $\tau \dot{I}_j = -I_j + \sum_i E_{ji} \delta_{Ty_j}$
Soma $\langle \delta_{\text{so}_j} \rangle_t = \lceil 0, \alpha_j(I_j + \beta_j) \rceil$
$D_{ji}\delta_{\text{so}_i}$
$T_{ji}\delta_{y_i}$
$E_{ji}\delta_{Ty_j}$
I_j

Figure 1.2: Operations and signal representations for an NEF network with two pools of neurons connected by decode-transform-encode weights. Three neurons emit spikes, modeled as unit delta trains (δ_{so_j}), with rates ($\langle \delta_{\text{so}_j} \rangle_t$) instantaneously determined by their input. The deltas are then scaled by their decode weights (different line thicknesses; positive orange, negative blue; zero gray), and superposed to realize weighted summation, producing a train of deltas with inhomogeneous areas. Transform weights and summations work the same way as with the decode. After being projected to the next pool's neurons by the encode weights, synapses low-pass filter each neuron's weighted deltas, forming continuous time, analog-value currents (I_j). Because the summation conserves deltas and fanout in weight matrices replicates them, the original 9 deltas turn into 45 deltas by the time they reach the 5 synapses.

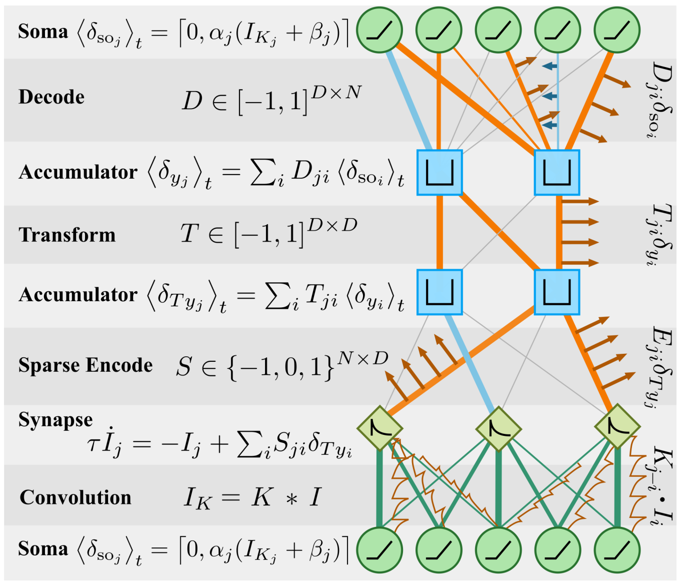
approximation, but spike noise will increase if the synaptic time-constant must be reduced to accommodate the additional layers.

By grouping neurons into pools, transistor mismatch and temperature sensitivity may be abstracted away. Mismatch of neuron gains and biases is desirable (to some extent), leading to an inherent variety of basis functions. The temperature sensitivity of the neurons' gain and bias parameters and mismatch in synaptic time-constants is undesirable, but the framework has recently been extended to compensate for them at the pool level (Kauderer-Abrams et al., 2017; Voelker et al., 2017).

1.2 Mapping the NEF to Neuromorphic Hardware

To efficiently decode and encode in our mixed analog-digital substrate, we focused on sparsifying digital communication in both time and space. For temporal sparsity, we invented novel machinery for combining weighted delta trains, the *accumulator*, which reduces total delta counts through layers, achieving the same SNR at a lower output rate than prior approaches. For spatial sparsity, we represent encoders not as a dense matrix, but as a sparse set of digitally programmed locations in the 2D array of analog neurons, each assigned a particular preferred direction. The *diffusor,* a transistor-based implementation of a resistive mesh (mimicking dendritic trees), convolves the output of these *tap-points* with its kernel*,* realizing well-distributed preferred directions. Using accumulators for decoding and tap-points and diffusors for encoding supports the NEF's abstractions while improving Braindrop's energy-efficiency (Figure 1.3 on page 6).

This thesis begins with a thorough exploration of the Accumulator and Tap Point operators, and proceeds to describe the hardware that implements those operators. Chapter 2 describes how the accumulator is able to peform weighting on streams of spikes while acheiving output statistics that approach that of a uniform point-process as the magnitude of the weights is lowered. Practical considerations, such as the effect of having weights that mix positive and negative signs, are also explored. Chapter 3 describes how encoders are constructed using the combination of the sparse encode matrix and the diffusor's convolutinal kernel. The ability of such encoders to cover the space they encode uniformly is measured for varying dimensionality of the representation and numbers of tap points.



Soma $\langle\delta_{\mathrm{so}_j}\rangle_t = \lceil 0, \alpha_j(I_{K_j}+\beta_j)\rceil$
Decode $D \in [-1,1]^{D\times N}$
Accumulator $\langle\delta_{y_j}\rangle_t = \sum_i D_{ji}\langle\delta_{\mathrm{so}_i}\rangle_t$
Transform $T \in [-1,1]^{D\times D}$
Accumulator $\langle\delta_{Ty_j}\rangle_t = \sum_i T_{ji}\langle\delta_{y_i}\rangle_t$
Sparse Encode $S \in \{-1,0,1\}^{N\times D}$
Synapse $\tau\dot{I}_j = -I_j + \sum_i S_{ji}\delta_{Ty_i}$
Convolution $I_K = K * I$
Soma $\langle\delta_{\mathrm{so}_j}\rangle_t = \lceil 0, \alpha_j(I_{K_j}+\beta_j)\rceil$
$D_{ji}\delta_{\mathrm{so}_i}$
$T_{ji}\delta_{y_i}$
$E_{ji}\delta_{Ty_i}$
$K_{j-i} \bullet I_i$

Figure 1.3: Braindrop's computational model replaces Figure 1.2 on page 4's superposition with accumulators and accomplishes encoding with a sparse matrix followed by convolution. Three neurons' weighted delta trains (with areas 4/6, −1/6, and 5/6), enter the accumulator, which sums the delta's areas to produce an output stream of unit-area deltas. The transform's weight is 1. Instead of a single layer of encode weights, transform outputs are first sent through sparse encode weights, which specify sets of synapse tap-points to send each dimension's deltas to. After the synapse, the outputs of the tap-points are convolved before being delivered to the neurons. For this particular configuration, the 9 input spikes are thinned to 4 before being being broadcast by the sparse encoders, yielding 8 total spikes to the synapses instead of 45, while still reaching all neurons.

Chapter 4 describes the hardware modules implementing and supporting these operators at a high level. Architectural tradeoffs, in particular, the balance between density, efficiency and flexibility are discussed. Chapter 5 details the implemenation of Braindrop's digital datapath, including the process decompositions for all major hardware modules.

Chapter 2

The Accumulator: Event-Based Weighting

By replacing the ideal summation with the accumulator, Braindrop avoids an explosion of traffic and avoids hardware multipliers, while simplifying the analog synapse's circuit design. The accumulator accomplishes this by summing the rates of deltas instead of superposing the deltas directly. This is functionally equivalent to the ideal summation, since the NEF encodes the values of delta trains by their filtered rates. Operating on rates allows us to restrict the areas of each delta in the accumulator's output train to be $+1$ or -1 , encoding values by modulating only the rate and sign of the outputs.

For the usual case of weights smaller than one, the accumulator produces a lower-rate output stream, reducing traffic compared to superposition. Because superposition conserves spikes from input to output, in an event-based NEF implementation, the matrix multiply operations lead to an explosion of traffic because $O(D_{\text{in}})$ deltas entering a matrix will result in $O(D_{\text{in}}D_{\text{out}})$ deltas being output. This multiplication of traffic compounds with each additional weight matrix layer. When implementing a weight matrix, there is one accumulator associated with each output dimension. When a delta associated with a particular input dimension occurs, the weights connecting it to each of its outputs are looked up and sent to each output dimension's associated accumulator. For a N–D–D–N decode-transform-encode, $O(N)$ deltas from the neurons results in $O(N^2D^2)$ deltas delivered to the synapses (Figure 1.2 on page 4). In contrast, the accumulator yields $O(ND)$ deltas to the synapses of the equivalent network (Figure 1.3 on page 6). This scaling assumes that the accumulator's output rate is proportional to a single neuron's spike rate. In practice, this corresponds to the desired SNR of the output value, as we will discuss.

The accumulator output's unit-area deltas simplify analog synapse circuit designs. Implementing a synapse that takes in multilevel inputs requires a digital-to-analog converter, which is extremely costly in terms of area. For this reason, compact silicon-synapse circuit designs take in only unit-area deltas, with signs denoting excitatory and inhibitory inputs (Neckar et al., 2018). Therefore, streams of variable-area deltas must be converted back to a stream of unit-area deltas before being delivered to the synapses, which the accumulator accomplishes.

2.1 Accumulators: Efficient Decoding by Thinning

Mechanistically, the accumulator operates as a deterministic thinning process that yields less noisy outputs than prior probabilistic approaches for combining weighted streams of deltas. Traditionally, delta-train thinning-as-weighting for neuromorphic chips has been performed probabilistically as Bernoulli trials (Goldberg et al., 2001; Choudhary et al., 2012). Like the accumulator, this method yields unit-area delta trains, but their statistics are Poissonian: SNR scales as $\sqrt{\lambda}$, where λ is the rate of deltas. The accumulator's improved statistics arise from having a state variable (x in Alg. 2.1) that is used to space output deltas more evenly in time. Given a Poisson input, the accumulator outputs a delta trains whose SNR scaling lies between the Poisson processes's $\sqrt{\lambda}$ and a periodic processes's $1/\lambda$, approaching the latter as the weights decrease.

To build intuition for why the accumulator produces delta trains that approach a periodic process as the weight decreases, we consider the case of a single accumulator with a single Poisson input. This is equivalent to the case where the accumulator performs a decode from many neurons, each with the same weight. Given a constant input, neurons spike periodically, rather than with Poisson statistics, but as long as each neuron's spike period is somewhat greater than the synaptic time-constant, τ, the inter-delta invervals (ISIs), *within the scope of the filter window* are independently distributed $X_i \sim \text{Exponential}(\lambda)$. Given uniform weights for each delta, $w = {}^1\!/k$, the ISI of the accumulator's output is $Y = \sum_i^k X_i$.

Algorithm 2.1 Accumulator Update

```
Require: input $w \in [-1,1]$
  $x := x + w$
  if $x \geq 1$ then
      emit +1 output
      $x := x - 1$
  else if $x \leq -1$ then
      emit -1 output
      $x := x + 1$
  end if
```

The coefficient of variation of Y is

$$\mathrm{CV}(Y) = \frac{\sigma_Y}{E[Y]} = \frac{\sqrt{k\,\mathrm{var}(X)}}{k\,E[X]} = \frac{\sqrt{k/\lambda^2}}{k/\lambda} = \frac{1}{\sqrt{k}} = \sqrt{w}$$

The decline in $\mathrm{CV}(Y)$ with increasing k suggests that the smaller the weight, w, the less random the the ISI of the accumulator's output will be. Intuitively, by taking sums of increasing numbers of the Poisson process's ISIs, the accumulator produces an output that approaches the statistics of a periodic process of rate λ/k.

The accumulator decimates the input delta train to produce its outputs, performing the desired weighting and yielding an output that is more efficiently encoded than the input, preserving most of the input's SNR while using fewer deltas (Figure 2.1 on page 11). The accumulator is fed with deltas from an inhomogeneous Poisson process, modeling the superposed spikes of many neurons with time-varying output spike rates. Since the synapse filters the accumulator's output delta-train to drive the neurons, to quantify performance, we evaluate the $\mathrm{SNR} = E[X]/\sqrt{\mathrm{var}(X)}$ of the filtered waveform X. The Bernoulli trials method accomplishes the same weighting, but the resulting Poissonian outputs have relatively poor SNR.

The accumulator's output can have statistics ranging from Poisson to periodic, depending on the value of k. It can be shown (Fok et al., submitted for publication) that its filtered output's SNR, R_g, is related to the Poisson input's SNR, R_p, by

$$R_\mathrm{g}^2 = R_\mathrm{p}^2 \,/\, (1 + k^2/\,3R_\mathrm{p}^2) \tag{2.1}$$

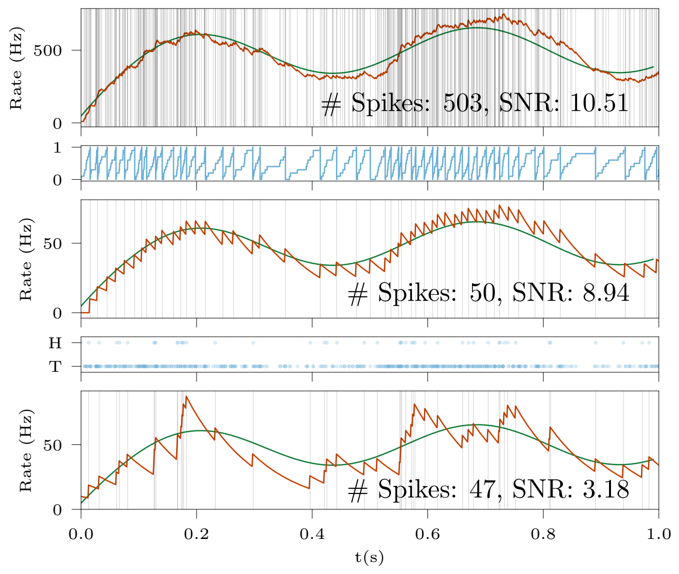


Figure 2.1: Accumulator operation. *Top*: Spikes (vertical lines) generated by an inhomogeneous Poisson process are filtered (orange); the ideal output is also shown (green). *Middle:* An accumulator fed with the same spikes ($w = 0.1$) yields a similar SNR. The accumulator state increments by w with each input spike and thresholds at 1, triggering an output spike (*first inset*). *Bottom*: A biased coin ($p = 0.1$) is flipped for each spike. When the coin returns heads (*second inset*), an output spike is generated. The reported SNRs are computed over a longer run of the same setup.

Using this formula, together with $R_\mathrm{p} = \sqrt{2\tau F_\mathrm{in}}$, where F_in is the rate of the Poisson process, we see that the for a given F_in, k may be increased for 1-2 decades without impacting the SNR (Figure 2.2 on page 13). At some point, however, further increases to k begin to degrade SNR. This degradation sets in when the SNR approaches that of a periodic process. This finding confirms our intuition that, for a high enough k, the accumulator will produce periodic statistics. In contrast, when $k = 1$—with the accumulator acting as a pass-through—the SNR matches a Poisson process's. Bernoulli weighting only produces Poisson outputs, always yielding lower SNR than the accumulator for the same F_out.

2.2 Practical Considerations for the Accumulator

To understand the dependence of energy consumed by the accumulator on k, we can study how, for a given SNR, the pre-and-post accumulator rates depend on k. By the thinning action of the accumulator, $F_\mathrm{out} = F_\mathrm{in}/k$. From Eq. 1, we derive $F_\mathrm{out} + F_\mathrm{in} = \frac{k+1}{k}\frac{1}{6}(3R_\mathrm{g}^2 + \sqrt{3R_\mathrm{g}^2(4k^2 + 3R_\mathrm{g}^2)})$, which we assume to be proportional to overall accumulator energy. Hence, for any desired output SNR, there is an optimal k that minimizes $F_\mathrm{in} + F_\mathrm{out}$ (Figure 2.3 on page 14). The flatter regions near each optimum corresponds to the vertical segments of the iso-F_in lines in Figure 2.2 on page 13, and the sloped sections to the right correspond to when k is made too large and SNR starts to be limited by the periodic process SNR limit.

All preceding analysis of the accumulator dealt with an idealized case where all neurons are assigned the same decoding weight; in practice, decoding weights not only vary in magnitude, but also in sign. This mixing has a negative impact on SNR, which can be studied through numerical experiments.

Our setup allows us to vary the amount of mixing, while controlling for the accumulator output rate. We constructed a 1-D population of 100 integrate-and-fire neurons with thresholds uniformly distributed in $[-1,1]$, and gains set so that each neuron's maximum firing rate was the same. We then trained this population for an increasingly difficult family of functions, each requiring more positive/negative mixing of weights than the last, solving the convex optimization $\mathrm{argmin}_\mathrm{d} \|A(x)d - F(x)\|_2^2 + \lambda \|d\|_2^2$ subject to $\forall_i\, |d_i| < 1$ (λ is a hyperparameter to prevent overfitting). The $|d_i| < 1$ constraint captures the requirement

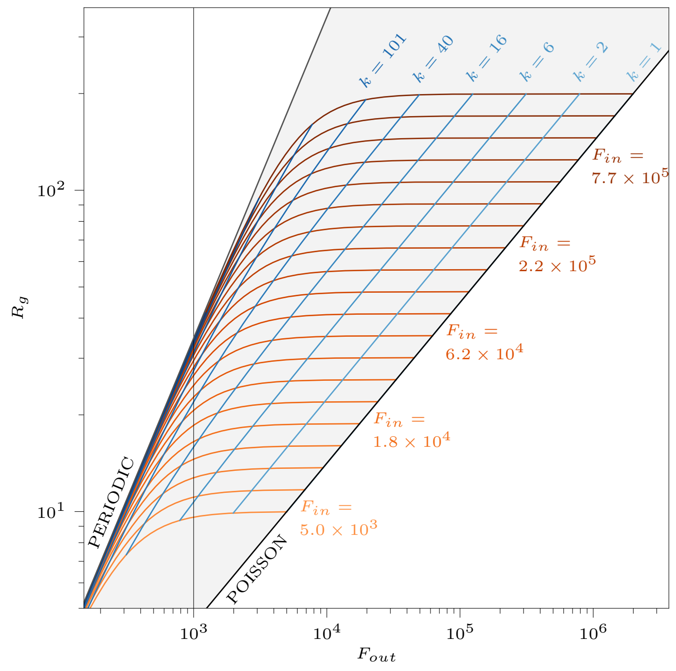


Figure 2.2: For a given Poisson input rate F_{in} (orange curves), as $k = {}^1\!/_w$ increases (blue curves), F_{out} drops while SNR remains constant, until a particular value of k is reached. The gray region demarcates feasible combinations of F_{out} and SNR, with point-process statistics transitioning from Poisson (slope ${}^1\!/_2$) to periodic (slope 1). The vertical line denotes when the accumulator is producing only ten spikes every τ seconds, below which it adds non-negligible jitter. The synapse itself causes delay on the order of τ: in response to a step change in input delta rate, after τ, a synapse's output will have risen $1 - 1/e \approx 63\%$ of the step. If the step is encoded in the accumulator's output, then the accumulator variable's initial state jitters the synapse's output waveform over a range of $1/F_{\text{out}}$, which should be kept small compared to the synapse's built-in order-τ delay.

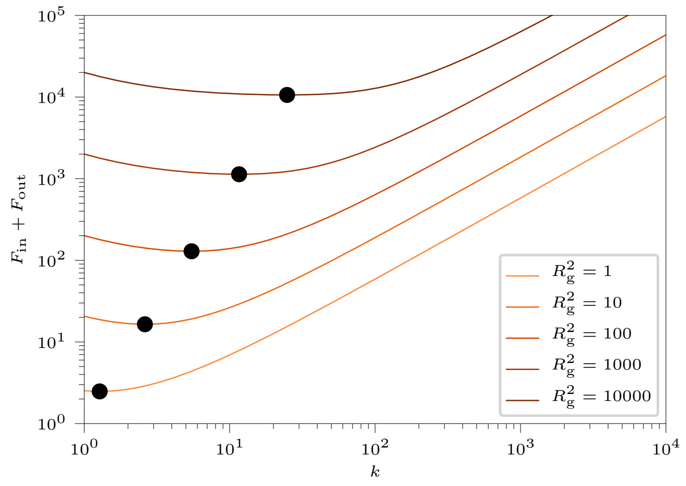


Figure 2.3: Optimizing dynamic power for a desired output SNR over k. Each line corresponds to a single output SNR, R_g^2. The dot corresponds to the $F_{in} + F_{out}$-optimizing k for that SNR.

that the accumulator only works with weights of magnitude less than 1. For our family of functions, we chose $F(x) = 1/2 + 1/2\sin(f\pi x)$; the larger f is, the more difficult the function is to approximate. We then considered the quality of the output at $x = 0$, a point chosen because output magnitude is constant and the error tends to be close to zero regardless of the choice of f. This is important because we want to consider an output of the same rate as we sweep f, to avoid confounding the cause of SNR changes. As f increases, the decoders become increasingly mixed-sign and SNR degrades accordingly (2.4).

Figure 2.5 shows how accumulator output SNR decays and how mixing increases with increasing function difficulty. Mixed-sign decoders immediately cause a decay in SNR. The flat region of the green curve shows that the output of the accumulator remains completely composed of positive events, for some time, even as the amount cancellation performed in the decode (the red line) increases. This cancellation is important because additional events sent downstream incur additional dynamic power and because the analog synapses which ultimately receive them must trade off bandwidth for dynamic range. (High gain between the synapse input and output is desirable, and gain is limited by the maximum spike rate the synapse can accept. Sending a mixture of positive and negative spikes, which contribute nothing to the signal but consume bandwidth, reduces the effective gain). In contrast, the

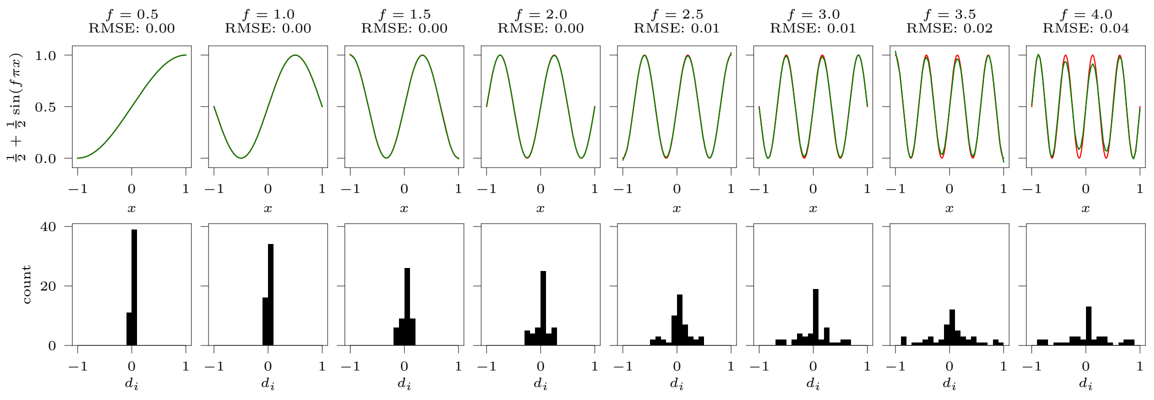


Figure 2.4: Degree of decode weight sign-mixing varying with function smoothness. The top row shows decode performance across the entire input range, reported as RMSE. The bottom row shows the decode weight distribution for neurons that have nonzero firing rate at $x = 0$.

Bernoulli trials weighting approach would display no such resilience, as any cancellation needed in the decoder would manifest proportionately in the output stream.

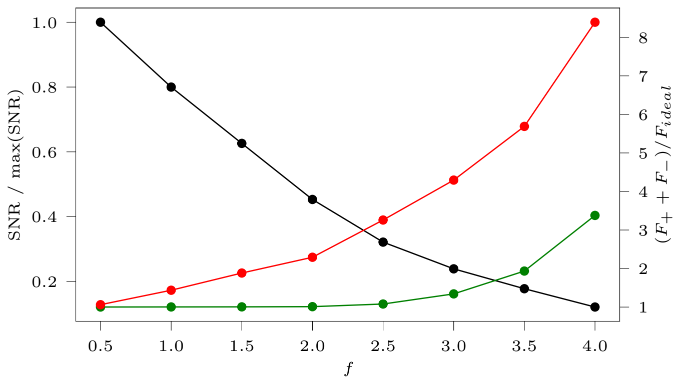


Figure 2.5: SNR decreasing and two measures of mixing increasing with f. The black line shows the decay of SNR. The red line is $A|d|$ and the green line is the sum of the positive and negative output spike rates from the accumulator.

Chapter 3

Tap Points: Encoding in Analog

Leveraging the inherent redundancy of NEF encoders, Braindrop uses the analog diffusor to efficiently fan out and mix outputs from a sparse set of *tap-points*[1]. In the NEF, the greatest fanout takes place during encoding because the encoders form an overcomplete basis for the input space. This overcompleteness motivates our encoder implementation using sparse tap-points and the diffusor: each neuron's resulting encoder is the summation of the *anchor encoders* of nearby tap-points, modulated by a weight that depends on the neuron's distance to those tap-points. Using this approach, it is possible to assign varied encoders to all neurons without specifying and implementing each one digitally, saving power by limiting digital fanout to the sparse tap point locations rather than to every neuron.

3.1 Tap-points and the Diffusor: Efficient Analog Communication and Computation

The diffusor is a resistive mesh implemented with transistors that sits between the synapse's outputs and the neuron's inputs, spreading each synapse's output currents among nearby neurons according to their distance from the synapse (Feinstein, 1988; Boahen and Andreou, 1992) (Figure 3.1 on page 18). The space-constant of this kernel is tunable by

[1]Meant to evoke the *taproot* of some plants, a thick central root from which smaller roots spread.

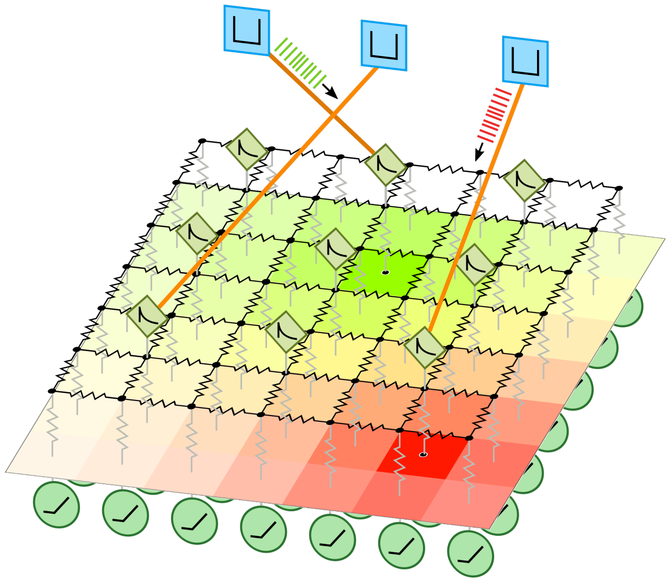


Figure 3.1: Diffusor and tap-point operation. Three accumulator buckets are shown feeding the synapse array, with two currently emitting outputs (red, green) and the third silent. The multicolored plane represents the currents delivered to each neuron at the corresponding diffusor output, colored according to what proportion of the input was from the red or green bucket and shaded according to the total input magnitude. Braindrop's resistive mesh is implemented as a hex-grid, which requires 50% more transistors but results in a more circular decay profile.

adjusting the gate biases of the transistors that form the mesh. Nominally, the diffusor implements a (2D) convolutional kernel on the synapse outputs and projects the results to the neuron inputs (Figure 1.3 on page 6). That the convolution takes place after the synapse is inconsequential: because of the synapse's linear nature, its operator could be swapped with the sparse encode (S, in Figure 1.3 on page 6) or the diffusor's convolution. From a set of tap-point locations $P_i = (x_i, y_i)$ with associated anchor encoders $C_i \in \mathbb{R}^D$ and (for space-constant γ) approximate diffusor decay profile $\hat{d}_\gamma(x)$, the encoder for neuron j at location $l_j = (x_j, y_j)$ is

$$e_j \approx \sum_i \hat{d}_\gamma(\|P_i - l_i\|_2) C_i \tag{3.1}$$

For low input-space dimensionality, only a handful of tap-points are needed to achieve an encoder distribution which covers the space well (Figure 3.2 on page 20). Ideally, as in the examples shown in the figure, anchor encoders are standard-basis vectors, because this takes advantage of the sparse encode operation, S. Alternatively, anchor encoders may be assigned arbitrarily using an additional transform.

The savings in fanout from tap-points becomes more limited as the number of dimensions is increased. The diffusor's action implies that the encoders of neighboring neurons will be similar (relatively small angles apart). The diffuser (as implemented, though not necessarily) is a 2D structure, implying that implementable encoders must be embeddable into a 2D metric space with little distortion. As the number of directions that need to represented increases exponentially with the number of dimensions, this becomes a more and more taxing constraint (Section 3.2 explores this in more detail).

Using the tap-points and diffuser to implement encoders is a desirable tradeoff. Encoders are typically chosen randomly, so the precise control of the original $\mathbb{R}^{N\times D}$ matrix is not missed. In exchange, we save a great deal of digital communication when encoding. For D of 1 to 3, it is possible to use a constant number of tap points to encode the input space nearly as uniformly as possible. For dimensions higher than three, we will see that the encoders are at least able to provide a modest constant-factor reduction in communication for a marginal degradation of performance.

3.2 Tap-point Encoders for $D > 3$

To study how generating encoders becomes more difficult as D increases, we must first quantify encoder coverage. Encoders that tile the surface of a D-sphere uniformly are generally desirable. One measure of coverage is the shape of the distribution of the closest encoder to a randomly chosen unit-vector on the surface of the D-sphere (Figure 3.3 on page 21). This distribution may be approximated by Monte-Carlo: generating a large number of random unit-vectors and finding each vector's largest inner-product among the normalized encoders. The inner product, proportional to the neuron's input, is a measure of closeness, equal to $\cos\theta$ for unit vectors, where θ is the angle separating the vectors.

Encoders of neurons far from tap-points tend to have small norms because of the fast decay of the diffusor kernel. This would manifest as a weak sensitivity to input for the associated neuron, but is mitigated by a tunable gain multiplier for each neuron, restoring

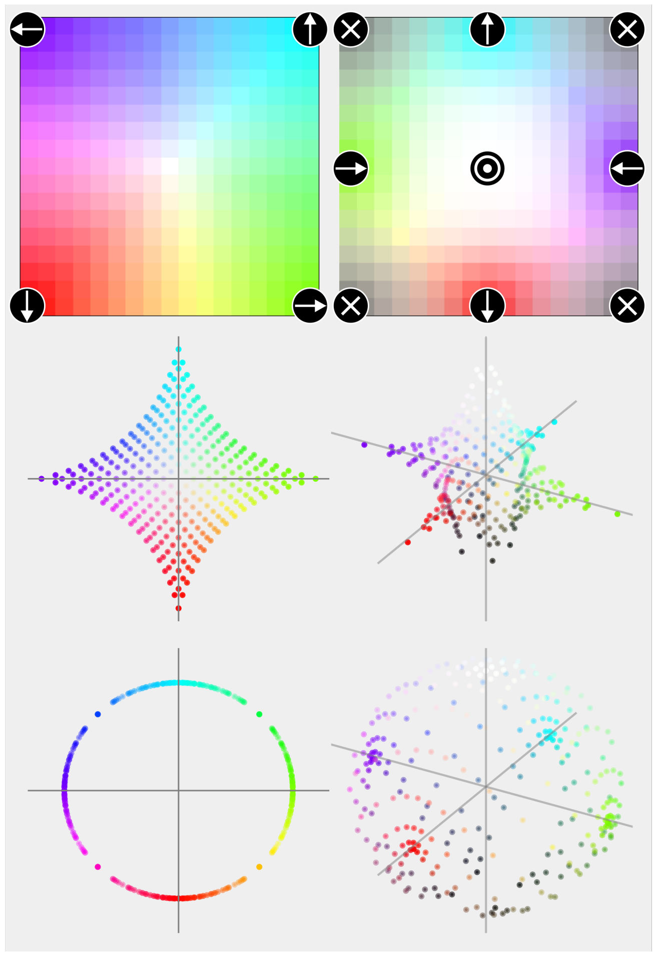


Figure 3.2: Encoding 2D (*left column*) and 3D (*right column*) input spaces with tap-points and the diffuser, for a 16×16 array of neurons, using standard-basis anchor encoders. *Top*: 4 and 9 tap-points are used (black dots, labeled with their anchor encoder) for 2D and 3D, respectively. For 2D, encoder direction maps to hue. For 3D, the first dimension maps to luminance (white to black) and the other two to hue. Shorter encoders are more transparent. *Middle*: Resulting encoders are plotted in the input spaces. *Bottom*: Encoders are normalized, showing that they achieve good radial coverage.

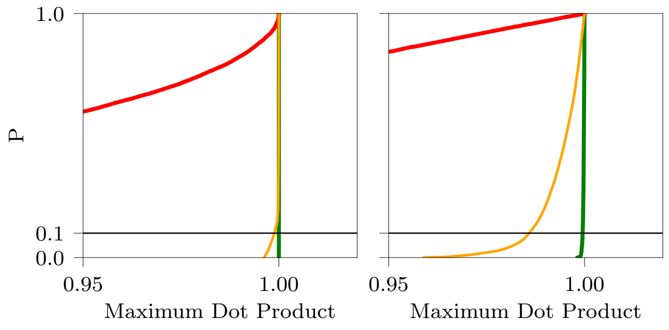


Figure 3.3: Encoder coverage CDFs for the encoders in Fig. 4.2, (orange). Red and green lines are two limiting cases: standard-basis encoders and encoders chosen uniformly randomly from the surface of the D-sphere. For 2D and 3D, only a few tap-points are needed to achieve very good coverage.

the encoder's length. When performing our normalization, we assume a dynamic range in this gain of 20, discarding any encoders less than $1/20^{\text{th}}$ the length of the longest encoder.

We measured coverage of tap-point encoders for increasing D, also varying the tap point density, ρ (Figure 3.4 on page 23). To reduce the CDF to a single number, we considered the 10^{th} percentile's performance, measuring the minimum $\cos\theta$ for 90% of the space.

Anchor encoders were selected greedily, picking encoders that were orthogonal to their neighbors. Tap-points were restricted to being fixed on a regular grid, evenly distributed in the neuron array (the diffusor's space-constant varied inversely to tap point spacing, $\gamma \sim \sqrt{1/\rho}$). We raster-scanned through the the tap points, assigning an anchor encoder to each, picking one that was orthogonal to its already-assigned neighbors. The size of the neighborhood considered increased with D (e.g. for $D = 3$, a tap point can be orthogonal to up to $D - 1 = 2$ neighbors, chosen as the left- and above-adjacent tap points. For $D = 4$, the tap point diagonally to the above-left can be added, and so on). Our rationale was to maximize mixing between anchor encoders. Adjacent anchors encoders that are aligned with each other produce similarly aligned encoders in the space between. These additional encoders will not contribute much to coverage. Conversely, the encoders that result for the neurons between orthogonal anchors span a 90° arc, boosting coverage.

Since there is randomness in the encoder generation process, and our greedy algorithm can reach some pathological cases (such as all the anchors being associated with a single quadrant), we generated several sets of encoders for each number of tap points and dimensions, and chose the best one. Following this, we performed a cross-validation, generating a second set of randomly generated unit-vectors to compute the reported performance.

We see that either approach can reduce digital communication substantially with only a marginal loss of performance for moderate ρ values. Performance is near-ideal even for a handful of tap-points for 2D and 3D (in fact, there is little benefit of using more than 4 or 9 tap points, respectively). This is likely because D-dimensional encoders sit on a $(D-1)$-dimensional surface, which maps perfectly to the diffusor's 2D metric space for up to 3 dimensions. For $D > 3$, about twice as many standard basis encoders are needed to perform as well as unrestricted anchor encoders.

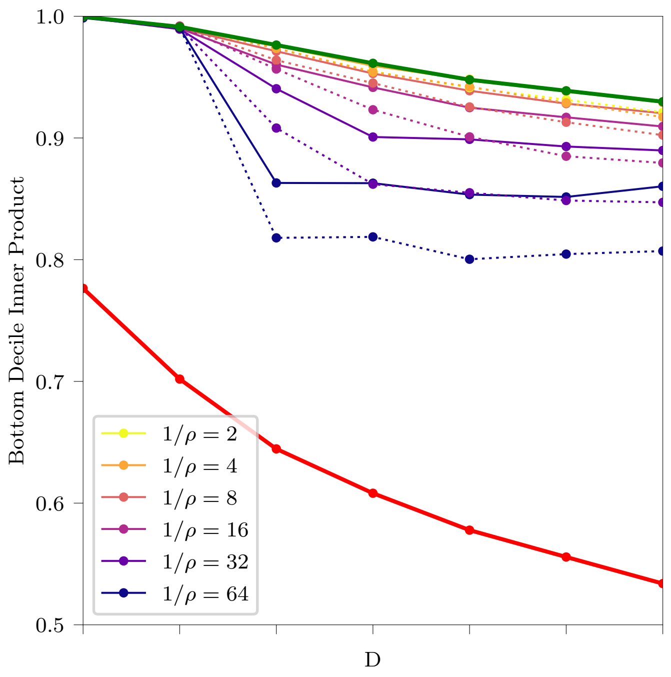


Figure 3.4: Tap-point encoders' performance as a function of the dimensionality of the representation (D) and the number of tap-points used (ρ per neuron). The green and red lines indicate two limiting cases: green is the performance of uniformly randomly distributed encoders, and red is the performance of the set of standard-basis vectors. The number of neurons is $N = 64 \cdot 2^D$ for all measurements, to preserve the performance of the uniform encoders limiting case. $1/\rho$ represents a factor by which digital communication is cut versus fully specifying encoders. Solid lines correspond to unrestricted anchor encoders, and dashed lines correspond to standard-basis anchor encoders.

Chapter 4

Braindrop Architecture

Braindrop is intended to be an instantiation of a single core of *Brainstorm*, a one-million-neuron system linking its cores together with an on-chip routing network. The original target application was Spaun (Eliasmith et al., 2012), which was used to guide the relative sizing of Braindrop's hardware modules. Some aspects of Braindrop's design, in particular the relatively complicated FIFO, were parametrized with the full, million-neuron system in mind.

This chapter begins by first describing the overall architecture, and proceeds by describing how we optimized the hardware to balance the various implementation costs. The parametrization of Braindrop is then reported, as well as the resulting layout areas for each module.

4.1 High-Level Architecture

In most cases, the mapping from the elements of Braindrop's computational model (see Figure 1.3 on page 6) to hardware modules is one-to-one (Figure 4.1 on page 26): *Synapses*, *Diffusor*, *Neurons*, and *Accumulator* units implement their named layers. An Accumulator is served by two memories, a large one to store the weights, and a small one to store the bucket states. An *Address-Event Representation Receiver* and *Transmitter* (AER:RX, AER:TX) *(Fok and Boahen, 2018) i*s responsible for demultiplexing digital input to the synapses and multiplexing spike outputs from the neurons, respectively. A *Pool Action*

Table (PAT) is responsible for dividing the neuron array into logical pools, transforming spikes (represented by address-events) into the addresses that point to the pool's decode weights and accumulator state variables in the accumulator's memories. A *Tag Action Table* (TAT) converts decoded or transformed quantities' delta trains (identified by *tags*) into one of three output types: different tags (optionally with global routes to other cores); synapse addresses with a polarity attached to each; or addresses of weights in accumulator's memories, used when implementing a transform.

The TAT enables compact storage of sparse connectivity schemes, with each input producing a series of *actions* drawn from the three output types. Spike outputs implement the sparse encode matrix, S. The zeros do not require any storage. Accumulator outputs enable transforms, yielding the same outputs as PAT entries, flexibly allocating transform matrices and buckets from the weight memory. Unlike the PAT, the TAT does not share a single entry among multiple inputs, but this is acceptable because there are far fewer multidimensional quantities than neurons. Tag fanout entries emit a series of tags with global route fields, which may point to other cores.

4.2 Optimizing Memory Utilization

In Braindrop, the organizations and sizes of memories associated with each resource class were carefully considered with respect to the size of the neuron array. We sought to avoid situations where either neurons or memories would be underutilized because of mismatches in the user's desired network topology and the physical constraints of the hardware.

By providing an additional layer of indirection, the PAT and TAT recoup efficiency with better effective resource utilization. The most area-expensive resources in the design are the analog neurons and the weight memory storing decode weights. Avoiding underutilization of these resources is a priority. When a neuron spikes, its spikes are always immediately decoded. The simplest, most efficient-up-front approach would be to assign each neuron a fixed range of memory addresses. However, in a large network, the D matrices have a wide variety of output dimensions. Fixing the memory address range allocated to each neuron implies a maximum output dimension for D, and underutilization for all pools that have fewer decoding dimensions. Allocating some neurons more memory than others would be

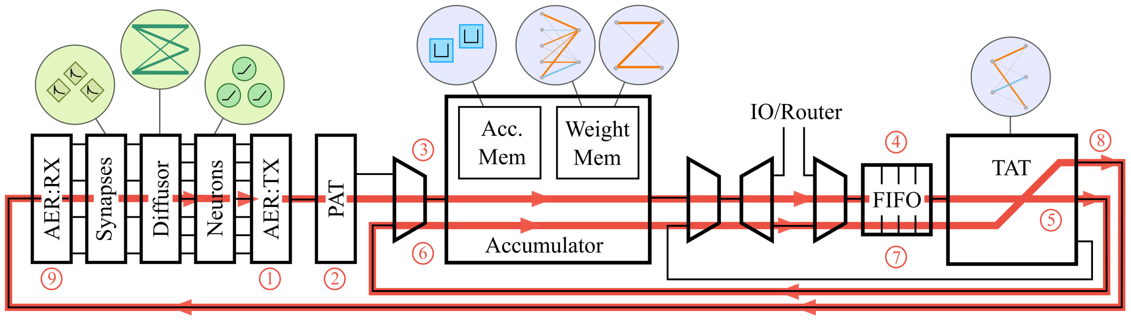


Figure 4.1: Block diagram of Braindrop's major components. The physical mapping of the decode-transform-encode operation (see Figure 1.3 on page 6) is illustrated. Circled objects from Figure 1.3 on page 6 illustrate where the resources each layer uses reside and the numbered red line shows the flow of traffic. **1**: Output spikes from the neurons are mapped into addresses by the AER transmitter. **2**: The address-events enter the PAT. For each pool, the PAT outputs the base address in the Accumulator of that pool's decode matrix, D, and the associated accumulator buckets. The address of neuron i's column of the matrix, D_i, is computed by concatenating the base address with i. **3**: The addresses enter the Accumulator unit, which performs the decode operation for each neuron by walking along its D_i. At each row j, that row's accumulator bucket state is updated with $w = D_{ij}$. Tags associated with each matrix row are emitted if the bucket's magnitude exceeds threshold. **4**: Tag streams representing decoded multidimensional vectors traverse the FIFO. **5**: The tag streams enter the TAT. In this case, each decoded quantity's tag is associated with a single set of Accumulator addresses, each pointing to a row of the transform matrix T and its buckets. **6**: The accumulator performs the transform, emitting tags associated with the multidimensional transform output. **7**: The transformed tag streams traverse the FIFO. **8**: In this case, the tags address locations in the TAT that store a list of tap-point addresses. Each tap point address will result in an address-event being sent to a particular synapse. Each entry gets a polarity, either flipping or preserving the sign of the delta, implementing -1 and $+1$ entries in S. **9**: Address-events enter the AER receiver and are delivered to their addressed synapses. The synapses filter the spike trains into currents, which are distributed by the diffusor to the neurons.

an improvement, but still imposes a maximum decoder dimension, and no distribution of allocations will work for all large networks.

The PAT consumes only marginal area, because neurons from the same pool may share entries (they all go to the same decoder matrix). In practice, implementing the PAT saves area because the much larger weight memory may now be arbitrarily allocated to the neurons. For the purposes of utilization, the PAT implies that the architect only has to worry about choosing the correct ratio of total neurons to total memory, instead of worrying about guessing the distribution of decoder dimensions.

The TAT negotiates the same tradeoffs for multidimensional quantities' delta trains, represented by tags. There is no canonical structure for the transformations between the output of the decoders and the input to the encoders, nor is there even a fixed ratio of the number of unique multidimensional quantities to the number of neurons. The size of the TAT's address-space implies a maximum number of tags for the core, but the address-space is freely divisible among those tags. Similar to the PAT, the TAT reduces the architect's job to balancing the number of tags to total complexity of the actions associated with those tags. These quantities should be considered with respect to the total number of neurons and amount of memory in the core. Since multidimensional quantities are few with respect to neurons, and area overhead is high for small memories, an attractive option is to make the TAT larger than the anticipated average need. This results in a marginal up-front increase in area, but is likely recouped by eliminating mapping constraints that could hurt neuron or weight memory utilization.

The segregation of the address-space with global routes and local tags saves power because most of the core operates using short addresses instead of long ones. As positioned, the FIFO and TAT only have entries for *local* tags addressing resources on the same core. The FIFO's size is $O(n \log n)$ and the TAT's size is $O(n)$, if n is the maximum number of local tags.

4.3 Optimizing Bandwidth Utilization

Because Braindrop's computations unfold in real-time with respect their description, throughput was only a concern around the synapses. The synapse has to serially generate pulses of multi-microsecond duration, and can potentially block faster types of traffic if they share multiplexed links. Because of the action of accumulators and tap-points, along with individual neuron spike rates being less than 1 KHz, other components did not have stringent throughput requirements. Except in a few pathological cases, latency was never a concern: everything digital happens orders of magnitude faster than the timescales at which neurons operate.

Core-to-core communication is optimized by segregating the high-bandwidth neuron-to-decoder spike traffic within individual cores, transmitting only multidimensional tag traffic inter-core as needed. The user controls the frequency of each tag stream based on their performance needs, with 1 kHz yielding SNRs between 10 and 200 depending on the target synapses' τ and the F_{in} of the population the spikes were decoded from. With far fewer multidimensional quantities than neurons, sending tags requires much less throughput than sending the raw spikes, justifying the tight binding of the neuron array, PAT, and accumulator.

4.4 Deadlock Prevention and Quality of Service Among Resource Classes

The FIFO implements deadlock prevention, providing a consumption channel with an overflow function. Since the frequency scales of each tag stream are set by the user, the peak traffic flowing across each link in the network and offered to each core's components is known at mapping time. Burstiness of tag streams is suppressed by the accumulator, so repeated overflow implies failure to map the network effectively. Overflow likelihood is further minimized with greater FIFO depth. Finally, because no one tag is critical to the computation, intermittent dropping of tags is merely detrimental to performance, rather than disastrous. Resource dependency cycles can exist even in single-core-only mappings

(Figure 4.1 on page 26), and the situation becomes much more complicated when mapping networks to many cores. The trivial implementation and minor computational consequences of dropping packets won out against the relative complexity of a lossless deadlock avoidance scheme (e.g., virtual channels).

Spikes entering the synapse's low-pass filter are approximated by a square pulse at least 50 μs wide, giving individual synapses throughput in the range of tens of kHz. The AER receiver is single-issue, so throughput is only maximized when cycling through all synapses round-robin. Repeated inputs to the same synapse will cause anything upstream to stall for microseconds.

The combined operation of the FIFO and TAT implement a round-robin protocol on synapse inputs, which is important to avoid pathologically low throughput of spikes sent to the neurons. The FIFO is primarily responsible for implementing the round-robin protocol: a circular buffer is maintained that keeps track of the set of unique tags in residence, and specifies the order that they shall be emitted in. An additional memory is used to store the number of each unique tag in the FIFO. Jitter on the order of a synaptic pulse width is possible in this scheme, because the offered throughput to the synapses must be less than their maximum throughput. This means that the FIFO will sometimes be empty, making it possible that the same tag can enter the FIFO (leading to the same synapse) before all the other tags (and their associated synapse inputs) have had a chance. In this situation, subsequent traffic will stall until the synapse which has just been targeted prematurely is no longer blocked. But in the intervening time, the FIFO will queue the inputs in an order that can be emitted as quickly as the AER receiver will allow, effectively catching up as quickly as possible. The TAT completes the round-robin with its assignment of tags to synapse inputs: as long as the same synapse is not used more than once as a tap-point, the round-robin order should allow the synapses to parallelize as much as possible.

Even when synapse throughput is maximized, putting all tag traffic in a single stream is problematic because the synapse will still periodically block other types of traffic. Suppose that all synapses are to be sent inputs, and all synapses operate at the same speed: the first set of inputs to each neuron goes in as fast as the AER receiver will allow, but the second set must wait for the first synapse to be able to accept a second input. Once this occurs, the remainder of the second set of will go in. Any other traffic being serviced by the same

TAT will also experience this stall while the first synapse becomes free. To prevent this, the synapse traffic can be treated as a separate resource class in the FIFO and TAT. At a minimum, this requires a separate set of buffers in the TAT, a separate circular buffer in the FIFO, and additional arbitration for shared logic and physical memories. Braindrop implements two TATs in parallel and effectively has two parallel FIFOs, except the FIFOs share a single physical memory for their circular buffers and a second physical memory for their tag counts (See Chapter 5 for further details).

4.5 Physical Implementation

Braindrop has 4096 neurons and 64KB weight memory, corresponding to sixteen 8-bit weights per neuron. We were accounting for an average decode dimension of 8, a figure close to Spaun's average, with generous additional space to implement transforms.

The PAT has 64 entries, dividing the neuron array into sixty-four 64-neuron sub-arrays. This division implies a pool size granularity of 64, slightly more than the NEF rule-of-thumb of 50 neurons per dimension for linear functions.

As discussed, the TAT and accumulator bucket memories were simply sized large enough to avoid mapping constraints, but not so large as to impact total area: there are 2048 total TAT entries and 1024 accumulator buckets. These are still relatively small memories, and total system area is relatively insensitive to changes made around these sizes.

Braindrop's digital logic was designed in a quasi-delay-insensitive (QDI) asynchronous style (Martin, 1993). Because digital computation is sparse in time, asynchronous digital logic's active-power–to-work-intensity proportionality is particularly desirable. Asynchronous circuits are completely idle when no inputs are offered, but run as fast as the transistors will allow on their arrival. This property is difficult to achieve in synchronous design.

Braindrop is implemented in a 28nm FDSOI process. The ability to reverse body-bias the transistors was essential to get the leakage as low as possible for the subthreshold analog circuits, but was also highly desirable for the digital logic, letting us trade off peak throughput for much lower digital leakage. Unfortunately, the foundry SRAM bitcell, which dominates digital transistor count, does not take advantage of this.

Component	Count	Unit Area (μm^2)	Total Size ($10^3\mu$m^2)	Percent of Total Area		
Neuron	4096	27.5	112.7	13.3	26.0	44.5
Synapse	1024	43.8	44.9	5.3		
DAC	12	5300	63.6	7.5		
AER	256	538	137.8	16.2	18.5	
Config Mem.	256	75.1	19.5	2.3		
Weight Mem.	1	0.637	334.0	39.3	39.3	55.5
Acc. Mem.	1	0.540	21.0	2.5	11.3	
PAT	1	2.344	3.0	0.4		
TAT	2	0.814	45.0	5.3		
FIFO Mem.	1	0.661	27.8	3.3		
Datapath	1	N/A	39.9	4.7	4.7	

Table 4.1: Areas of major Braindrop components (including unused space inside bounding boxes). Datapath memory unit areas are reported as area per bit.

The layout is roughly divided into digital and analog sections (Figure 4.2 on page 32 and Tab. 4.1). The overall area is dominated by the weight memory and the neuron array and associated DACs. As discussed, other memories were designed to be large enough that their associated resources were unlikely to cause mapping constraints, but not so large they they had a large impact on the total system area. Because no async-compatible memory IP is available from the foundry, we designed the entirety of Braindrop's memories (except for the bitcell). Due to time constraints, all of our memories involve some standard-cell layout for the peripheries, and have high overhead.

We optimized the number of gain and bias bits to implement for each neuron to maximize useful neuron yield (Neckar et al., 2018). For a fixed amount of area, more useful neurons result from implementing a small number of gain and bias bits, rather than by simply sizing up the circuits to reduce the mismatch. Gain and bias bits are stored in the config memory, a 128-bit SRAM in the 16-neuron tile. The memory only implements a write operation; an additional inverter drives a wire leading out of each cell to the analog circuits that use the stored value. The config memory also stores kill bits for the neurons and synapses, and cut bits that allow the diffusor to be broken at pool boundaries.

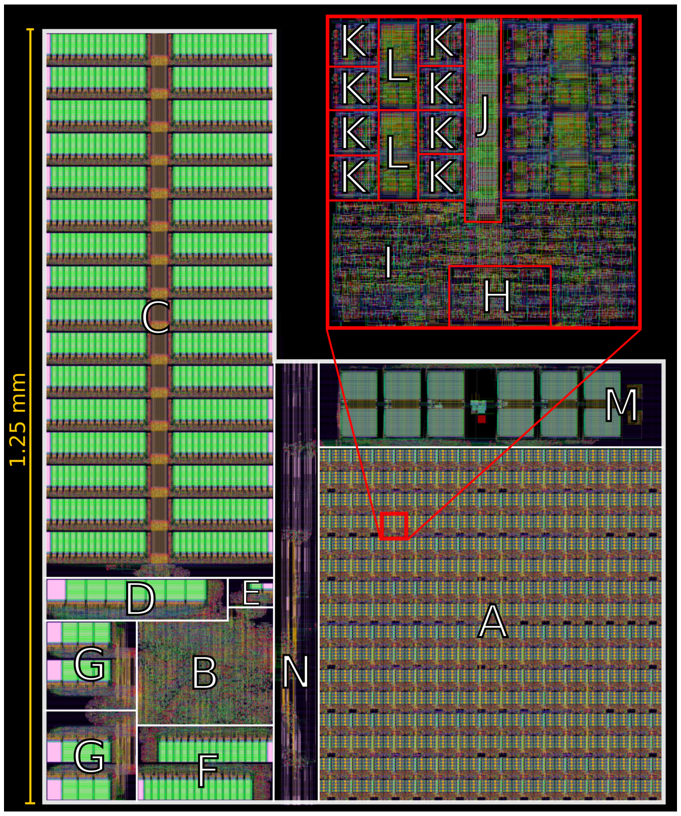


Figure 4.2: Layout of Braindrop. Red inset shows detail of 16-neuron tile. **A**: 4096-neuron array, **B**: digital datapath, **C**: weight memory, **D**: accumulator memory, **E**: PAT memory, **F**: FIFO memory, **G**: TAT memories, **H**: AER tree logic, **I**: AER leaf logic, **J**: config SRAM, **K**: neuron, **L**: synapse, **M**: 12 DACs and 2 ADCs, **N**: routing between neuron array and datapath and to IO.

4.6 Software Stack

To go from an abstract description of a computation to its implementation on mismatched, temperature-variable hardware, Braindrop is supported by extensive synthesis software. Nengo (Bekolay et al., 2014) is a software environment for implementing NEF. It consists of a *frontend*, which provides a set of objects (pools, nodes, connections, etc.) that the user describes their computation with, and a *backend* that provides a means to implement that computation. It also provides a GUI to make interaction with the frontend more user-friendly. The backend for Braindrop interfaces with Braindrop's driver software, which provides its own set of objects, nearly isomorphic to the hardware itself, and provides methods to communicate with and control an attached Braindrop chip. Once the network has been configured using the software stack, Braindrop may be detached from the PC. This gives access to the efficiency of Braindrop's hardware using an abstract language-–the same implementation-agnostic Nengo interface that is used for many available backends.

Chapter 5

Implementation of Braindrop's Datapath

The digital datapath is responsible primarily for performing the digital weighting of spikes with the accumulator, for enabling the specification of tap points, and for directing and buffering digital traffic. The datapath is composed of four primary blocks: the Pool Action Table (PAT), the Accumulator unit, the FIFO, and the Tag Action Table (TAT) (Figure 5.1 on page 35). This chapter begins by describing each of these modules' behaviors at a high level, and proceeds to detail each one's implementation in complete detail.

5.1 Functional Descriptions of Primary Components

The Communicating Hardware Processes (CHP) (Martin, 1993) descriptions for the main digital datapath components are reproduced below. This specifies the behavior of each block at a high level.

5.1.1 Pool Action Table

The PAT divides the neuron array into a number of sub-arrays of fixed size, and stores one set of base Accumulator addresses for each sub-pool. The neuron address is divided into a sub-array index and neuron index within that pool. The PAT is indexed with the sub-array

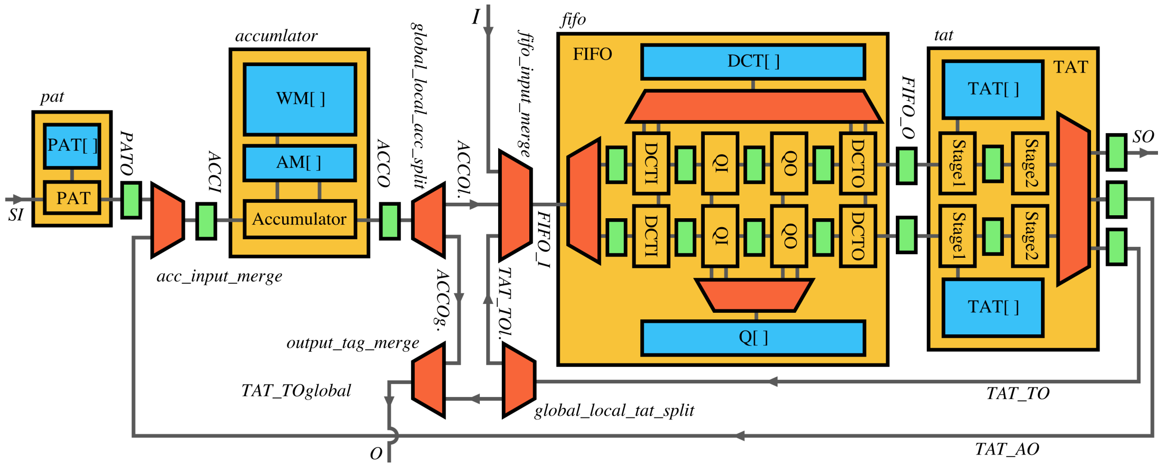


Figure 5.1: Braindrop's datapath, accepting input channels I (tag inputs) and SI (spike inputs from the neuron array) and driving output channels O (tag outputs) and SO (spike outputs to the neuron array). Major pipeline stages (yellow boxes) and their memories (blue) are separated by PCFBs (green). Traffic is directed between components by splits and arbited merges (orange). Coming out of each half of the TAT, the traffic is split by destination type. Both halves' traffic for each type is then merged. This is simplified in the schematic.

index looking up an address to the accumulator bucket state memory, and a column address to the weight memory, corresponding to the first row of the D matrix. The neuron index is used as a row address to the weight memory, indexing a particular column of the D matrix.

```
$Pool\ \ Action\ \ Table(I,\ \ O) \equiv$
    $*[I?(subarray, neuron);$
      $(wma_{y,base}, wma_x, ama) := \text{PAT}[subarray];$
      $O!((wma_{y,base}, neuron), wma_x, ama)]$
```

5.1.2 Accumulator

Starting from the input addresses, the accumulator unit "walks" along the rows of both memories in parallel. Each read yields the inputs to the accumulator operation for one row of the D matrix: from the weight memory, the decode weight; from the accumulator bucket memory the buckets states, bucket threshold values, the tags ids to emit, and the stop bit that signals the end of the row. The resulting bucket state is written back to the memory. For a fixed output magnitude, weight magnitude will vary inversely to the number of neurons in a pool. To preserve dynamic range, the scale of weights in a given row is adjustable. This is implemented with a variable accumulator threshold value, stored as powers of two of the maximum weight value that can be represented. The operation is performed in one's complement to make the threshold detection and correction efficient. The accumulator emits signed tags, representing the signed deltas associated with a particular dimension of the decode. Each decode output dimension will have a different *tag_out* value to identify its delta train uniquely.

```
$Accumulator(I,O) \equiv$
        $stop := 0$
        $*[stop \longrightarrow$
          $I?(wma, ama, sign), stop := 0$
         $\square \neg stop \longrightarrow$
          $d_{ij} := \mathrm{WM}[wma], (v, thr, stop) := \mathrm{AM}[ama];$
          $[sign = 1 \longrightarrow v' := v + d_{ij} \square sign = -1 \longrightarrow v' := v \downarrow d_{ij}];$
          $trigger := v'_{thr} \otimes \mathrm{sign}(v');$
          $[trigger \longrightarrow O!(tag\_out, \mathrm{sign}(v')), v'_{thr} := \neg v'_{thr}];$
          $\mathrm{AM}[ama].v := v, wma := wma + 1, ama := ama + 1$
         $]$
```

5.1.3 Tag Action Table

The TAT emits a series of outputs for each single input tag. It is divided into two stages. The first stage reads from the memory until the stop bit is encountered. The second stage interprets each output from the first. Inputs to the TAT and FIFO are associated with a count. The signed output of the accumulator corresponds to counts of $+1$ and -1, but the magnitude of these counts can grow if tags pile up in the FIFO (the FIFO has only a single entry for each unique tag, and keeps track of the total count of those tags). Since the Synapse and Accumulator do not take multiple-count inputs, those outputs also reinsert the input tag in the FIFO with the count decremented, by making a tag output with a route of 0. Their logics are:

```
$TAT(I, AO, SO, TO) \equiv$
        $stop := 0$
        $*[stop \longrightarrow I?(tag, ct); addr := tag$
         $[]\neg stop \longrightarrow (stop, type, data) := TAT[addr];$
            $S!(type, data, ct), addr := addr + 1]$

        $*[S?(type, data, ct);$
           $[type = \textbf{GlobalTag} \longrightarrow$
              $(route, tag) := data; TO!(route, tag, ct)$
           $[]type = \textbf{SynapseSpike} \longrightarrow$
              $(sign_0, addr_0, sign_1, addr_1) := data;$
              $(SO!(sign_0 \otimes \text{sign}(ct), addr_0),$
               $SO!(sign_1 \otimes \text{sign}(ct), addr_1));$
              $TO!(0, tag, ct - \text{sign}(ct))$
           $[]type = \textbf{AccumulatorInput} \longrightarrow$
              $(wma_x, wma_y, ama) := data;$
              $AO!(wma_x, wma_y, ama, \text{sign}(ct)),$
              $TO!(0, tag, c - \text{sign}(ct))]]$
```

To avoid the synapse blocking other traffic, Braindrop actually implements two separate TAT units, one addressing the lower half of the range of tag values (assigned to the synapse-bound spike outputs) and one addressing the upper half (assigned to the other other outputs). The TAT may be pipelined, doing the memory operations in the first stage, and handling the different output types in the second.

5.1.4 FIFO

The FIFO achieves great depth by storing not only the sequence of inputs, but their total counts. The FIFO is implemented with two memories. The first memory is direct-mapped by the tag inputs, keeping track of the *dirty* bit for each tag (whether it is in residence), and the count of those tags in the FIFO. The second memory keeps track of the order of tag inputs, implemented as a circular buffer. Inputs that would cause the count of a tag to exceed the max value will instead simply cause the value to saturate, providing a consumption channel. The user is notified of this event by an overflow warning.

The overall logic is as follows. DCT and Q are the dirty/count and circular buffer memories, respectively. I and O are input and output channels. The $OVFLW$ communication is assumed to consume, signaling an overflow to the user. Variables $head$ and $tail$ manage the circular buffer: $head = tail$ implies an empty queue.

```
$FIFO(I,O,OVFLW) \equiv$
    $head := 0, tail := 0$
    $*[\bar{I} \longrightarrow$
        $I?(tag, ct_{in});$
        $(d, ct_{curr}) := \mathrm{DCT}[tag];$
        $ct_{new} := ct_{curr} + ct_{in};$
        $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW]$
        $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW]$
        $\mathrm{DCT}[tag] := (1, ct_{new}),$
        $[\neg d \longrightarrow \mathrm{Q}[tail] := tag, tail := tail + 1]$
     $| \overline{O} \wedge (head \neq tail) \longrightarrow$
        $tag := \mathrm{Q}[head]; head := head + 1,$
        $(d, ct_{curr}) := \mathrm{DCT}[tag];$
        $\mathrm{DCT}[tag] := (0, 0),$
        $[ct_{curr} \neq 0 \longrightarrow O!(tag, ct_{curr})]$
     $]$
```

This may be decomposed into two parallel processes, each one servicing a memory. P is the channel between the *input* sides of the DCT and Q processes, used when *putting* something into the queue. G is the channel between the *output* sides of the Q and DCT processes, used when *getting* something from the queue. G implements an exchange communication: DCT cannot simply trigger on the output requesting data because it does not know whether the queue is empty. The logic for the DCT process is:

```
$DCTFIFO(I,P,G,OVFLW) \equiv$
        $*[\bar{I} \longrightarrow I?(tag, ct_{in});$
            $(d, ct_{curr}) := \mathrm{DCT}[tag];$
            $ct_{new} := ct_{curr} + ct_{in};$
            $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW]$
            $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW]$
            $\mathrm{DCT}[tag] := (1, ct_{new}), [\neg d \longrightarrow P!tag]$
        $|\overline{G} \longrightarrow G?tag;$
            $(d, ct_{curr}) := \mathrm{DCT}[tag];$
            $\mathrm{DCT}[tag] := (0,0),$
            $[ct_{curr} \neq 0 \longrightarrow O!(tag, ct_{curr}])$
        $]$
```

The logic for the Q stage, implementing the circular buffer is:

```
$QFIFO(P,G) \equiv$
        $head := 0, tail := 0$
        $*[\overline{P} \longrightarrow P?tag, \mathrm{Q}[tail] := tag, tail := tail + 1$
          $|\overline{G} \wedge head \neq tail \longrightarrow G!\mathrm{Q}[head]; head := head + 1$
        $]$
```

Braindrop's design adds an additional layer of complexity, to avoid the synapse blocking other traffic types. There are two copies of the above processes, addressing the lower and upper halves of the tag address range (synapse and non-synapse traffic). Physically, there is only one DCT memory and one Q memory. Each memory arbitrates access between its two subscribers. This was done to avoid the area overhead of 4 smaller memories compared to 2 larger ones.

5.2 Top-Level Datapath Composition

Adding buffering between components to facilitate pipelining, and adding splits and merges where necessary, we obtain the following compositional process for the datapath (Figure 5.1 on page 35):

$BraindropDatapath(I, O, SI, SO, OFVLW) \equiv$

PAT$(SI, PATO)$ *pat*
PCFB$(PATO, PATObuf)$ *pat_pcfb*

ArbMerge$(PATObuf, TAT_AObuf, ACCI)$ *acc_input_merge*
PCFB$(ACCI, ACCIbuf)$ *acc_input_pcfb*

Accumulator$(ACCIbuf, ACCO)$ *accumulator*
PCFB$(ACCO, ACCObuf)$ *acc_output_pcfb*

Split$(ACCObuf, ACCObuf.global_route = 0, ACCOlocal, ACCOglobal)$ *global_local_acc_split*

ArbMerge3$(I, ACClocal, TAT_TOlocal, FIFOin)$ *fifo_input_merge*

FIFO$(FIFO_I, FIFO_O, OVFLW)$ *fifo*
PCFB$(FIFO_O, FIFO_Obuf)$ *fifo_out_pcfb*

TAT$(FIFO_Obuf, TAT_AO, TAT_SO, TAT_TO)$ *tat*
PCFB(TAT_AO, TAT_AObuf) *tat_acc_out_pcfb*
PCFB(TAT_SO, SO) *tat_spike_out_pcfb*
PCFB(TAT_TO, TAT_TObuf) *tat_tag_out_pcfb*

Split$(TAT_TObuf, TAT_TObuf.global_route = 0, TAT_TOlocal, TAT_TOglobal)$ *global_local_tat_*

ArbMerge$(ACCOglobal, TAT_TOglobal, O)$ *output_tag_merge*

Above, PCFB, ArbMerge, and Split circuits have been introduced as well. The PCFB provides a full unit of slack, breaking the circuits before and after into pipeline stages. The Split drives the input port's data to one of its output ports, determined by a function on the input data. The ArbMerge collects multiple inputs, serializing their data onto the same output port, using arbiter circuits to resolve collisions.

Programming and diagnostic ports have been omitted from this and the following descriptions. Large trees of Splits and ArbMerges, called the Horn and Funnel, respectively, are used to program memories, send commands, and receive diagnostic outputs from the various components. Programming commands are merged into each memory without arbitration, so they can only be issued when the component is not under load.

To shut off the flow of traffic without obliterating the network state, Valve circuits are inserted at various points in the datapath. There are Valves before the PAT, before the FIFO, and after the FIFO. Each Valve has two bits of configuration, the first controls whether the inputs are forwarded to the output. The second controls whether the input is sent into the Funnel for diagnostic purposes.

5.3 Synthesis of the Datapath

5.3.1 The QDI Async Synthesis Process

According to the standard practices of QDI asynchronous design, Braindrop's logic was synthesized in a largely manual fashion, using control-data decomposition. There are no widely-used tools that allow for synthesis of asynchronous digital logic from a behavioral description.

Control-data decomposition is an async design approach that aims to makes synthesis tractable by divorcing the circuitry that determines the sequencing of operations from the circuitry that does the computation. First, the processes's desired operation is analyzed at an abstract level and the sequencing requirements are identified. The original process is then decomposed into a set of control and data processes. Each resulting data process should contain an operation which will ultimately be implemented as part of the same handshake. The control processes enforce the sequencing of these handshakes.

5.3.2 CHP and HSE Syntax Conventions

For the most part, the CHP, HSE, and PRS conventions used in this document are consistent with those in Martin (1993) and Manohar (2009), but some extensions are made to enhance clarity. Notably, we add explicit syntax for modular composition of processes, and explicit data types for HSE and PRS.

5.3.2.1 CHP Conventions

We add some stylistic rules to CHP. Capital variables refer to channels (e.g. A), and lower-case variables refer to data elements. Roman characters refer to indexable memory arrays. Parameters are in boldface.

We also introduce a syntax for CHP that allows for composition via instantiation of process instances:

$$\text{ProcessType}(\textit{port\_a},\textit{port\_b},\ldots)\ \textit{instance\_name}$$

The following example illustrates this syntax in use. MainProcess take in an input on channel I, input_sub_proc (of type InputSubProcType, which must be described elsewhere) outputs something on channel A (internal to MainProcess), MainProcess transforms the data on A and inputs the result to output_sub_proc (of type OutputSubProcType), which finally drives an output to O:

$MainProcess(I,O) \equiv$

$$\text{InputSubProcType}(I,A)\ \textit{input\_sub\_proc\_instance}$$
$$\text{OutputSubProcType}(B,O)\ \textit{output\_sub\_proc\_instance}$$
$$*[A?x; y := \text{exciting\_function}(x); B!y];$$

This compositional syntax can also be employed for HSE and PRS.

The bullet operator, "•", is somewhat imprecise, implying overlapping communications. It is often used in control-data decomposition in the data processes. In this document, it is meant to imply one of two mappings to HSE:

$BulletProc(L,R) \equiv$

$$*[L \bullet R]$$

CHP $\triangleright$ HSE

$\mathit{BulletProc}(\overleftrightarrow{\mathrm{ed}}\ L,R) \equiv$

usually for linking a control channel to its data process

$*[L.d\uparrow;\ [\neg L.e];R.d\uparrow;\ [\neg R.e];L.d\downarrow;\ [L.e];R.d\downarrow;\ [R.e]]$

OR, usually for synchronizing parallel operations

$*[L.d\uparrow,R.d\uparrow;\ [\neg L.e \wedge \neg R.e];L.d\downarrow,R.d\downarrow;\ [L.e \wedge R.e]]$

5.3.2.2 HSE Conventions

In HSE, data types (which are usually implicit) are explicitly defined in each program's definition. The data type is not repeated for adjacent port names of the same type. Channels are denoted with bidirectional arrows over the type name (e.g. $\overleftrightarrow{someChannelType}$). The following types are commonly used:

- bool: a single-bit boolean
- 1of2: a dual-rail variable with $.t$ and $.f$ members. Data valid '0' is encoded with $.f = 1$ and $.t = 0$, Data valid '1' is encoded with $.t = 1$ and $.f = 0$. The neutral state is encoded with $.t = 0$ and $.f = 0$.
- 1of**N**: The **N**-bit generalization of dual-rail variable, with members $.d_i$, $i \in [0, \mathbf{N}-1]$. A value n is encoded with $d_n = 1$ and $d_{i \neq n} = 0$. The neutral state is encoded with $d_i = 0$ for all i.
- $\overleftrightarrow{\mathrm{ed}}$: a data-less channel with $.d$ (usually semantically equivalent to *request*) and $.e$ (usually semantically equivalent to *inverted acknowledge*) members.
- $\overleftrightarrow{\mathrm{e}(1\,\mathrm{of}\,2)}$: a channel passing a dual-rail variable
- $\overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\,\mathrm{of}\,2)}$: a channel passing an array of **N** dual-rail variables, contained by the array $.b$.

In general, an HSE or PRS program is defined:

$\mathit{SomeProcessName}(\overleftrightarrow{\mathrm{someChannelType}}\ \mathit{CHANNELNAME},\ \mathrm{someDataType}\ \mathit{dataname},\ \ldots) \equiv$

$*[$*process specification goes here*$]$

Channel port names are presented in uppercase (possibly with lowercase suffixes, e.g., $CHANNELbuf$). Single-direction (non-channel) port names are presented in lowercase (e.g., $somedataline$)

Each process resets to the beginning of its program (to the right of $*[$), unless otherwise noted with $\star$.

5.3.2.3 PRS Conventions

The only additional PRS notation is that combinational rules are written using the $\Longrightarrow$ symbol. $a \Longrightarrow b\downarrow$ implies both $a \longrightarrow b\downarrow$ as well as $\bar{a} \longrightarrow b\uparrow$.

5.3.3 Ubiquitous Circuits

Before presenting the decomposition of each datapath component, we will present some simple circuits that are reused throughout the design. This also gives an opportunity to demonstrate our unique syntactic conventions in a simple setting.

5.3.3.1 The Simplest Control Circuit, $*[L;R]$

This is probably the most elementary control circuit, used to implement two non-overlapping operations (Manohar, 2009). The following circuits both have the CHP $*[L,R]$, but have different HSE reshufflings. They are named accordingly

$LaRa(L,R) \equiv$

$$*[L;R]$$

CHP $\triangleright$ HSE

$LpRa(\overleftrightarrow{\mathrm{ed}}\ L,R) \equiv$ *"L passive, R active"*

$$*[[L.e];a\downarrow;L.d\uparrow;[\neg L.e];$$
$$R.d\uparrow;[\neg R.e];a\uparrow;R.d\downarrow;[R.e];L.d\downarrow]$$

$LaRa(\overleftrightarrow{\mathrm{ed}}\ L,R) \equiv$ *"L active, R active"*

$$*[L.d\uparrow;[\neg L.e];a\downarrow;L.d\downarrow;[L.e];$$
$$R.d\uparrow;[\neg R.e];a\uparrow;R.d\downarrow;[R.e]]$$

HSE $\triangleright$ PRS

$LpRa(\overset{\leftrightarrow}{\text{ed}}\ L,R) \equiv$ "*L passive, R active*"

$$\neg a \ \vee\ \neg R.e \ \longrightarrow\ L.d\uparrow$$
$$a \ \wedge\ R.e \ \longrightarrow\ L.d\downarrow$$

$$\_sReset \ \wedge\ L.e \ \longrightarrow\ a\downarrow$$
$$\neg\_sReset \ \vee\ \neg R.e \ \longrightarrow\ a\uparrow$$

$$\neg a \ \wedge\ \neg L.e \ \longrightarrow\ R.d\uparrow$$
$$a \ \vee\ L.e \ \longrightarrow\ R.d\downarrow$$

$LaRa(\overset{\leftrightarrow}{\text{ed}}\ L,R) \equiv$ "*L active, R active*"

$$\neg\_sReset \ \vee\ \neg a \ \vee\ \neg R.e \ \longrightarrow\ \_Ld\uparrow$$
$$\_sReset \ \wedge\ a \ \wedge\ R.e \ \longrightarrow\ \_Ld\downarrow$$
$$\_Ld \ \Longrightarrow\ L.d\downarrow$$

$$L.e \ \Longrightarrow\ \_Le\downarrow$$
$$\_sReset \ \wedge\ \_Le \ \longrightarrow\ a\downarrow$$
$$\neg\_sReset \ \vee\ \neg R.e \ \longrightarrow\ a\uparrow$$

$$\neg a \ \wedge\ \neg\_Le \ \longrightarrow\ R.d\uparrow$$
$$a \ \vee\ \_Le \ \longrightarrow\ R.d\downarrow$$

5.3.3.2 The WhileLoop, $*[stop \longrightarrow \star A[]\neg stop \longrightarrow B?stop]$

Quite often in Braindrop, an operation is executed as an initialization action, followed by repeated execution of another action, until some stop condition is reached (e.g., read sequentially from a memory, performing some action, until a stop bit is read). The WhileLoop process describes the sequencing of such a procedure. A synchronizes with the initialization and B synchronizes with the loop operation, which repeats until the $stop$ variable passed on B is true. The process has then been returned to the initial state, needing another loop initialization action to continue. The first version is compiled with a A passive (intuitively, the process is triggered by the environment) and B active (the process triggers the loop actions).

```
$WhileLoop(A,B) \equiv$
    $*[stop \longrightarrow \star A \,\square\, \neg stop \longrightarrow B?stop]$
```

CHP $\triangleright$ HSE

```
$WhileLoop(\overleftrightarrow{\text{ed}}\ A,\ \overleftrightarrow{\text{e(1of2)}}\ B) \equiv$
    $*[stop \longrightarrow$
        $[\neg A.d]\,;A.e\uparrow;\star stop\downarrow;\ [A.d]\,;A.e\downarrow$
     $\square \neg stop \longrightarrow$
        $B.e\uparrow;\ [v(B.d)]\,;stop := B.t;B.e\downarrow;\ [n(B.d)]$
     $]$
```

HSE $\triangleright$ PRS

```
$WhileLoop(\overleftrightarrow{\text{ed}}\ A,\ \overleftrightarrow{\text{e(1of2)}}\ B) \equiv$
    $\neg\_pReset \ \vee\ \neg B.t \ \wedge\ \neg\_stop \ \wedge\ \neg A.d \ \longrightarrow\ A.e\uparrow$
    $\_stop \ \wedge\ A.d \ \longrightarrow\ A.e\downarrow$

    $\neg sReset \ \wedge\ \neg\_Ae \ \longrightarrow\ \_stop\uparrow$
    $sReset \ \vee\ \_Ae \ \wedge\ B.t \ \longrightarrow\ \_stop\downarrow$

    $\neg B.f \ \wedge\ \neg\_\_stop \ \wedge\ \neg A.e \ \longrightarrow\ B.e\uparrow$
    $B.f \ \vee\ \_\_stop \ \vee\ A.e \ \longrightarrow\ B.e\downarrow$

    $\_stop \ \Longrightarrow\ \_\_stop\downarrow$
    $A.e \ \Longrightarrow\ \_Ae\downarrow$
```

The following version is instead compiled for an active A:

```
$*[stop \longrightarrow$
    $\star A.d\uparrow; [\neg A.e]; stop\downarrow; A.d\downarrow; [A.e]$
 $[]\neg stop \longrightarrow$
    $B.e\uparrow; [v(B.d)]; stop := B.t; B.e\downarrow; [n(B.d)]$
 $]$
```

HSE ▷ PRS

$\neg sReset \ \wedge \ \neg\_stop \ \wedge \ \neg B.t \ \longrightarrow \ A.d\uparrow$

$sReset \ \vee \ \_stop \ \vee \ B.t \ \longrightarrow \ A.d\downarrow$

$\neg\_\_Ae \ \longrightarrow \ \_stop\uparrow$

$pReset \ \vee \ \_\_Ae \ \wedge \ B.t \ \longrightarrow \ \_stop\downarrow$

$\neg\_\_stop \ \wedge \ \neg\_Ae \ \wedge \ \neg B.f \ \longrightarrow \ B.e\uparrow$

$\_\_stop \ \vee \ \_Ae \ \vee \ B.f \ \longrightarrow \ B.e\downarrow$

$\_stop \ \Longrightarrow \ \_\_stop\downarrow$

$A.e \ \Longrightarrow \ \_Ae\downarrow$

$\_Ae \ \Longrightarrow \ \_\_Ae\downarrow$

5.3.3.3 Composable Control, $*[T \longrightarrow L; R; T]$

The TarrowLaRaT process can be used to insert arbitrarily long sequences into other control processes. This circuit passively waits to be triggered by the T communication, then makes sequential L and R communications. Once those are complete, it finishes the T communication, signaling T's sender that L and R have completed.

$TarrowLaRaT(T, \ L, \ R) \equiv$

$$*[\overline{T} \longrightarrow L; R; T]$$

CHP ▷ HSE

$TarrowLaRaT(\overleftrightarrow{\text{ed}}\ T,\ \ L,\ \ R) \equiv$

$$\begin{aligned}&*[\,[T.d]\,;L.d\uparrow;\,[\neg L.e]\,;a\downarrow;L.d\downarrow;\,[L.e]\,;\\ &\quad R.d\uparrow;\,[\neg R.e]\,;T.e\downarrow;\,[\neg T.d]\,;a\uparrow;R.d\downarrow;\,[R.e]\,;T.e\uparrow]\,;\end{aligned}$$

HSE $\triangleright$ PRS

$TarrowLaRaT(\overleftrightarrow{\text{ed}}\ T,\ \ L,\ \ R) \equiv$

$$\begin{aligned}&L.e \implies \_Le\downarrow\\ &\_Ld \implies L.d\downarrow\\ \\ &T.d \ \wedge \ \ a \longrightarrow \_Ld\downarrow\\ &\neg T.d \ \vee \ \neg a \longrightarrow \_Ld\uparrow\\ \\ &\neg a \ \wedge \ \neg\_Le \longrightarrow R.d\uparrow\\ &a \ \vee \ \_Le \longrightarrow R.d\downarrow\\ \\ &\neg\_sReset \ \vee \ \neg T.d \longrightarrow a\uparrow\\ &\_sReset \ \wedge \ \_Le \longrightarrow a\downarrow\\ \\ &T.e \ = \ R.e;\end{aligned}$$

By constructing trees of this process, it is possible to make arbitrary control processes of the form $*[T \longrightarrow X_1;X_2;...;X_N;T]$. The T communication could be matched with another control processes's communication, effectively replacing the single communication in the parent process with a sequence of communications. For example, if I wanted the process $*[stop \longrightarrow A;B;C;D[]\neg stop \longrightarrow E?stop]$, I could make $*[T \longrightarrow A;B;C;D;T]$ out of three LarrowLaRaTs arranged in a tree ($*[T \longrightarrow T';T'';T] \parallel *[T' \longrightarrow A;B;T'] \parallel *[T'' \longrightarrow C;D;T'']$), and compose it with $*[stop \longrightarrow T[]\neg stop \longrightarrow E?stop]$, (which is a WhileLoop).

5.3.3.4 Logic Trees

HSE for a N-input ANDs, ORs, and C-elements, which may be implemented as trees:

$ANDN(\text{bool}{\times}\mathbf{N}\ in,\ \ \text{bool out}) \equiv$

$$*[\{\wedge : i\ \ in_i\}; out{\uparrow}; \neg\{\wedge : i\ \ in_i\}; out{\downarrow}]$$

$ORN(\text{bool}{\times}\mathbf{N}\ in,\ \ \text{bool out}) \equiv$

$$*[\{\vee : i\ \ in_i\}; out{\uparrow}; \neg\{\vee : i\ \ \neg in_i\}; out{\downarrow}]$$

$CN(\text{bool}{\times}\mathbf{N}\ in,\ \ \text{bool out}) \equiv$

$$*[\{\wedge : i\ \ in_i\}; out{\uparrow}; \{\wedge : i\ \ \neg in_i\}; out{\downarrow}]$$

5.3.3.5 Plain Register

Because it is a fundamental circuit that forms the basis for the IncrementingRegister (below), this fundamental circuit (Manohar, 2009) is reproduced here. This also an opportunity to demonstrate compositional syntax.

$Reg(W,R) \equiv$

$$*[\overline{W} \longrightarrow W?x \,\square\, \overline{R} \longrightarrow R!x]$$

CHP $\triangleright$ HSE

$Reg(\overleftrightarrow{\text{e}\mathbf{N}{\times}(1\text{of}2)}\ W,R) \equiv$

$$\begin{aligned}*[&v(W.b) \longrightarrow x := W.b; S.e{\downarrow};\ [n(W.b)]\,; W.e{\uparrow}\\ &\square R.e \longrightarrow R.b := x;\ [\neg R.e]\,; R.b{\downarrow}]\end{aligned}$$

HSE $\triangleright$ HSE

```
$RegCell(\overleftrightarrow{\mathrm{e(1of2)}}\ W,R) \equiv$
        $*[\overline{W.t \vee W.f} \longrightarrow$
            $[W.t \wedge x.f \longrightarrow x.t\uparrow; x.f\downarrow$
            $[]W.f \wedge x.t \longrightarrow x.f\uparrow; x.t\downarrow];$
             $[W.t \wedge x.t \vee W.f \wedge x.f]; W.e\downarrow; [\neg W.t \wedge \neg W.f]; W.e\uparrow]$
          $[]\overline{R.e} \longrightarrow R.b := x; [\neg R.e]; R.b\downarrow$
         $]$
```

```
$Reg(\overleftrightarrow{\mathrm{e}\mathbf{N}\times\mathrm{(1of2)}}\ W,R) \equiv$
        *declare internal variables*
        $\overleftrightarrow{\mathrm{e(1of2)}}\ Ws[\mathbf{N}]$
        $\overleftrightarrow{\mathrm{e(1of2)}}\ Rs[\mathbf{N}]$
        $\mathrm{RegCell}(Ws[:], Rs[:])\ Regs[\mathbf{N}]$

        *break out R*
        $\{:i:\mathbf{N}:\ Rs[i].e = R.e\}$
        $\{:i:\mathbf{N}:\ Rs[i].d = R.b_i.d\}$

        *break out W*
        $\mathrm{CN}(Ws[:].e, W.e)\ Wcomp$
        $\{:i:\mathbf{N}:\ Ws[i].d = W.b_i.d\}$
```

The PRS for the RegCell is notably clever in how it handles the write operation. The basic design resets to an undefined state. This behavior can be modified by adding reset transistors to the NANDs that controls $x.t$ and $x.f$.

HSE $\triangleright$ PRS

$RegCell(\overleftrightarrow{\mathrm{e(1of2)}}\ W,\ \overleftrightarrow{\mathrm{e(1of2)}}\ R) \equiv$

$$\neg W.f \ \wedge \ \neg x.f \ \implies \ x.t{\uparrow}$$
$$\neg W.t \ \wedge \ \neg x.t \ \implies \ x.f{\uparrow}$$

$$W.t \ \wedge \ x.t \ \vee \ W.f \ \wedge \ x.f \ \longrightarrow \ \_We{\downarrow}$$
$$\neg W.t \ \wedge \ \neg W.f \ \longrightarrow \ \_We{\uparrow}$$
$$\_We \ \implies \ W.e{\downarrow}$$

$$R.e \ \wedge \ x.t \ \implies \ \_Rt{\downarrow}$$
$$R.e \ \wedge \ x.f \ \implies \ \_Rf{\downarrow}$$
$$\_Rt \ \implies \ R.t{\downarrow}$$
$$\_Rf \ \implies \ R.f{\downarrow}$$

5.3.3.6 Incrementing Register

An incrementing register could be implemented with a pair of registers, an adder, and a $*[L,R]$ control circuit, but it is more area-efficient and faster to build an array of custom cells that do the modification in-place.

The write operation is straightforward, and functions almost identically to the plain register cell's write. The increment is facilitated by each cell's *ci*, *co*, and *a* ports. If a cell's *ci* is raised, and the stored value is 0, the value flips to 1 and the cell raises its *a*. If the stored value is 1, it flips to 0 and the cell raises its *co*. Each cell's *co* is the next cell's *ci*, and the LSB cell's *ci* is wired to $I.d$. $I.e$ is raised when one of the *a*s, or the MSB's *co* (denoting an overflow) is raised. The carry only propagates until it encounters a 0 bit, so the operation time is $O(\log N)$ where N is the number of bits.

$IncReg(W,R,I) \equiv$

```
*[$\overline{W} \longrightarrow W?x$
  $[] \overline{R} \longrightarrow R!x$
  $[] \overline{I} \longrightarrow x := x+1; I$
 ]
```

CHP $\triangleright$ HSE

```
$IncReg(\overleftrightarrow{\mathrm{e}\mathbf{N}\times(\mathrm{1of2})}\ W,\ \ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(\mathrm{1of2})}\ R,\ \ \overleftrightarrow{\mathrm{ed}}\ I) \equiv$
        $*[v(W.b) \longrightarrow x := W.b; W.e\downarrow; [n(W.b)]; W.e\uparrow$
          $[]R.e \longrightarrow R.b := x; [n(R.e)]; R.b\downarrow$
          $[]I.d \longrightarrow x := x+1; I.e\downarrow; [\neg I.d]; I.e\uparrow$
         $]$
```

HSE $\triangleright$ HSE

```
$IncRegCell(\text{bool } ci,\ \text{bool } co,\ \text{bool } a,\ \overleftrightarrow{\text{e(1of2)}}\ W,\ R) \equiv$
    $*[\overline{W.t \vee W.f} \longrightarrow$
        $[W.t \wedge x.f \longrightarrow x.t\uparrow; x.f\downarrow$
        $[]W.f \wedge x.t \longrightarrow x.f\uparrow; x.t\downarrow];$
          $[W.t \wedge x.t \vee W.f \wedge x.f]; W.e\downarrow; [\neg W.t \wedge \neg W.f]; W.e\uparrow]$
      $[]ci \wedge x.t \longrightarrow co\uparrow; (x.t\downarrow; x.f\uparrow), [\neg ci]; co\downarrow$
      $[]ci \wedge x.f \longrightarrow a\uparrow; (x.f\downarrow; x.t\uparrow), [\neg ci]; a\downarrow$
      $[]R.e \wedge x.t \longrightarrow R.t\uparrow; [\neg R.e]; R.t\downarrow$
      $[]R.e \wedge x.f \longrightarrow R.f\uparrow; [\neg R.e]; R.f\downarrow$
     ]
```

```
$IncReg(\overleftrightarrow{\text{e}\mathbf{N}\times\text{(1of2)}}\ W,\ \overleftrightarrow{\text{e}\mathbf{N}\times\text{(1of2)}}\ R,\ \overleftrightarrow{\text{ed}}\ I) \equiv$
    $\overleftrightarrow{\text{e(1of2)}}\ Ws[\mathbf{N}]$
    $\overleftrightarrow{\text{e(1of2)}}\ Rs[\mathbf{N}]$
    $\text{bool}\times\mathbf{N}\uparrow 1\ c$
    $\text{bool}\times\mathbf{N}\uparrow 1\ a$
    $\text{IncRegCell}(c[0:\downarrow 1], c[1:], a[:\downarrow 1], Ws[:], Rs[:])\ Regs[\mathbf{N}]$

    break out R
    $\{:i:\mathbf{N}:\ Rs[i].e = R.e\}$
    $\{:i:\mathbf{N}:\ Rs[i].d = R.b_i.d\}$

    break out W
    $\text{CN}(Ws[:].e, W.e)\ Wcomp$
    $\{:i:\mathbf{N}:\ Ws[i].d = W.b_i.d\}$

    connect carries, I.d is LSB ci
    $c[0] = I.d$
    MSB co functions like an a, could be used to detect overflow
    $a[\downarrow 1] = c[\downarrow 1]$

    collect acks with an NORN
    $\text{NORN}(a, Ie)\ inc\_completion$
```

HSE ▷ PRS

$IncRegCell(\text{bool } ci, \;\; \text{bool } co, \;\; \text{bool } a, \;\; \overleftrightarrow{\text{e(1of2)}}\; W,R) \equiv$

$$\begin{aligned}
&ci \;\wedge\; \_x.f \;\wedge\; \_a \;\longrightarrow\; \_o\downarrow\\
&\neg\_pReset \;\vee\; \neg ci \;\wedge\; \neg\_x.f \;\longrightarrow\; \_o\uparrow\\
&\_o \;\Longrightarrow\; o\downarrow
\end{aligned}$$

$$\begin{aligned}
&ci \;\wedge\; \_x.t \;\wedge\; \_co \;\longrightarrow\; \_a\downarrow\\
&\neg\_pReset \;\vee\; \neg ci \;\wedge\; \neg\_x.t \;\longrightarrow\; \_a\uparrow\\
&\_a \;\Longrightarrow\; a\downarrow
\end{aligned}$$

$$\begin{aligned}
&W.t \;\Longrightarrow\; \_W.t\downarrow\\
&W.f \;\Longrightarrow\; \_W.f\downarrow
\end{aligned}$$

$$\begin{aligned}
&\_W.f \;\wedge\; \_co \;\wedge\; \_x.f \;\Longrightarrow\; \_x.t\downarrow\\
&\_W.t \;\wedge\; \_a \;\wedge\; \_x.t \;\Longrightarrow\; \_x.f\downarrow
\end{aligned}$$

$$\begin{aligned}
&\neg\_W.t \;\wedge\; \neg\_x.t \;\vee\; \neg\_W.f \;\wedge\; \neg\_x.f \;\longrightarrow\; \_W.e\uparrow\\
&\_W.t \;\wedge\; \_W.f \;\longrightarrow\; \_W.e\downarrow\\
&\_W.e \;=>\; W.e\downarrow
\end{aligned}$$

R is implement like the plain register, with AND gates

5.3.3.7 PCFB

This is a standard buffer circuit, implementing a full unit of slack. It is often used to divide logic into pipeline stages. This, and other slack-providing cells are studied in detail in Lines (1998). The logic for a single cell is derived:

$PCFB(L,R) \equiv$

$*[I?x; O!x]$ *this is somewhat vacuous*

CHP ▷ HSE

$$
\begin{aligned}
&PCFB(\overleftrightarrow{\mathrm{e(1of2)}}\ L,R) \equiv \\
&\qquad *[\,[R.e]\,;\,[L.t \longrightarrow R.t{\uparrow} \,[\!]\, L.f \longrightarrow R.f{\uparrow}]\,;L.e{\downarrow};en{\downarrow}; \\
&\qquad\quad (\,[\neg L.t \wedge \neg L.f]\,;L.e{\uparrow}), \\
&\qquad\quad (\,[\neg R.e]\,;R.t{\downarrow},R.f{\downarrow});en{\uparrow}]
\end{aligned}
$$

HSE ▷ PRS

$PCFB(\overleftrightarrow{\mathrm{e}(1\mathrm{of}2)}\ L,R) \equiv$

$$\neg\_pReset \;\vee\; \neg\_\_en \;\wedge\; \neg R.e \;\longrightarrow\; \_Rt\uparrow$$
$$\_\_en \;\wedge\; R.e \;\wedge\; L.t \;\longrightarrow\; \_Rt\downarrow$$

$$\neg\_pReset \;\vee\; \neg\_\_en \;\wedge\; \neg R.e \;\longrightarrow\; \_Rf\uparrow$$
$$\_\_en \;\wedge\; R.e \;\wedge\; L.f \;\longrightarrow\; \_Rf\downarrow$$

$$\neg\_pReset \;\vee\; \neg\_\_en \;\wedge\; \neg L.t \;\wedge\; \neg L.f \;\longrightarrow\; \_\_Le\uparrow$$
$$\_\_en \;\wedge\; vR \;\longrightarrow\; \_\_Le\downarrow$$

$$\neg\_Rt \;\vee\; \neg\_Rf \;\longrightarrow\; vR\uparrow$$
$$\_Rt \;\wedge\; \_Rf \;\longrightarrow\; vR\downarrow$$

$$\neg sReset \;\wedge\; \neg\_Le \;\wedge\; \neg vR \;\longrightarrow\; en\uparrow$$
$$sReset \;\vee\; \_Le \;\longrightarrow\; en\downarrow$$

__en is generated to avoid having to size en's drivers too large

$$en \;\Longrightarrow\; \_en\downarrow$$
$$\_en \;\Longrightarrow\; \_\_en\downarrow$$

$$\_Rt \;\Longrightarrow\; R.t\downarrow$$
$$\_Rf \;\Longrightarrow\; R.f\downarrow$$

$$\_\_Le \;\Longrightarrow\; \_Le\downarrow$$
$$\_Le \;\Longrightarrow\; L.e\downarrow$$

Multi-bit PCFBs can be constructed by arraying the bitcell and adding a completion tree for the input's enable. Multiple units of slack may be created by putting PCFBs in sequence (it is not necessary to have the completion tree at each stage).

5.3.3.8 Arbiter

Quite often, we would like to build circuits that receive inputs on multiple channels that could potentially arrive simultaneously. The Arbiter circuit allows us to resolve these collisions and present mutually exclusive, sequenced inputs to the circuit.

```
$Arbiter(A,B) \equiv$
    $*[\overline{A} \longrightarrow A$
      $|\overline{B} \longrightarrow B$
     ]
```

CHP ▷ HSE

```
$Arbiter(\text{bool } a,b,u,v) \equiv$
    $(a,u)$ and $(b,v)$ may be thought of as forming channels
    $*[a \longrightarrow u\uparrow; [\neg a]; u\downarrow$
      $|b \longrightarrow v\uparrow; [\neg b]; v\downarrow$
     ]
```

The Arbiter is implemented as a pair of cross-coupled NAND gates. If a and b transition to V_{dd} simultaneously, the outputs of the NANDs, u and v, will both drop to around V_{inv}, before the metastability resolves and one side eventually wins, continuing to V_{ss} while the other returns to V_{dd} (Manohar, 2009). As long as the winning input is held high, the output will remain steady regardless of the behavior of the losing input. The incomplete transition of the loser is masked by a filter circuit. The filter is a set of cross-coupled inverters, where each inverter takes as its input one NAND's output, and uses the *other* NAND's output to drive its V_{dd} terminal. Internally, the u and v nodes still exhibit bumps, but the u and v outputs are clean, because u/v will not go high until v/u returns to V_{dd}.

5.3.3.9 Splits and Merges

Splits and merges are commonly used in composing modules to form more complicated processes, directing the flow of data about the circuit. Based on some condition, the Split takes the input and directs it to one of the outputs. The merge (which can exist in arbitered or un-arbitered forms) simply drives any input it receives to its output.

$$\mathit{Split}(I, C, O_0, O_1) \equiv$$
$$*[C?c; [\neg c \longrightarrow O_0!I? [] c \longrightarrow O_1!I?]]$$

$$\mathit{ExclusiveMerge}(I_0, I_1, O) \equiv$$
$$*[\overline{I_0} \longrightarrow O!I_0? [] \overline{I_1} \longrightarrow O!I_1?]$$

$$\mathit{ArbMerge}(I_0, I_1, O) \equiv$$
$$*[\overline{I_0} \longrightarrow O!I_0? \mid \overline{I_1} \longrightarrow O!I_1?]$$

CHP $\triangleright$ HSE

```
$Split(\overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ I,\ \ \text{1of2}\ c,\ \ \overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ O_0, O_1) \equiv$
        $c\ \ is\ \ assumed\ \ to\ \ cycle\ \ with\ \ I's\ \ data$
        $*[c.f \longrightarrow [v(I.b)]; O_0.b := I.b;$
            $[\neg O.e]; I.e\!\downarrow;$
            $[\neg c.f \wedge n(I.b)]; O_0.b\!\downarrow;$
            $[O.e]; I.e\!\uparrow$
         $[]c.t \longrightarrow$
            $\ldots\ \ like\ \ c.f$
         $]$

$ExclusiveMerge(\overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ I_0, I_1, O) \equiv$
        $*[v(I_0.b) \longrightarrow O.b := I.b;\ [\neg O.e]; I.e\!\downarrow;\ [n(I_0.b)]; O.b\!\downarrow;\ [O.e]; I.e\!\uparrow$
         $[]v(I_1.b) \longrightarrow\ \ \ldots\ \ like\ \ v(I_0.b)]$

$ArbitedMerge(\overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ I_0, I_1, O) \equiv$
        $\text{Arbiter}(a, b, u, v)\ arb$
        $*[v(I_0.b[0]) \longrightarrow a\!\uparrow;\ [b]; O.b := I_0.b;$
            $[\neg O.e]; I.e\!\downarrow;$
            $[n(I_0.b)]; a\!\downarrow;\ [\neg b]; O.b\!\downarrow;$
            $[O.e]; I.e\!\uparrow$
         $[]v(I_1.b[0]) \longrightarrow \ldots\ \ like\ \ v(I_0.b[0])$
         $]$
```

The implementation of each circuit is straightforward. The Split is implemented with one length-**N** array of C-elements per output. One input of each C-element is driven by an input bit, and the other is driven by either $c.t$ or $c.f$. The ExclusiveMerge is simply an array of OR gates for the data lines and C-elements to produce $I_0.e$ and $I_1.e$ (each fed by the validity check for one bit of that side's data and $O.e$).The ArbMerge uses an arbiter fed by $v(I_0.b[0])$ and $v(I_1.b[0])$. For each input, the data lines are gated by an array of C-elements is driven on one side by the input data lines and on the other by the output of the arbiter.

The outputs of the C-element arrays are merged to drive the output. *I.e* is computed as in the input merge.

5.3.4 SRAM Memories

Memory access was expected to dominate Braindrop's dynamic power consumption, so special interfaces were designed for the logic's sequential access patterns. Prioritizing energy over throughput, a single-word-granular hierarchical wordline was used, dividing each bank into columns. The exact number of entries needed is read out sequentially. The resulting three interfaces support the following commands:

1. RW: *read, write*
2. RI: *set address, read-increment address, write-increment address*
3. RMW: *set address, read-write, increment* address

RW interfaces are used for the PAT and FIFO memories. The RI and RMW interfaces support Braindrop's sequential accesses, as performed by the TAT and Accumulator, respectively. For RI, the TAT sets the base address once per set of sequential accesses and uses the read-increment interface several times to read entries sequentially. For RMW, the Accumulator sets the base address then makes interleaved requests to read-write (modifying the read data before writing it back) and increment. The read-write is performed as a single operation: the wordline is raised only once.

Originally, the addressing operations were meant to be implemented at a low level to reduce the overhead of each sequential access. The set address command would decode to a set of one-hot shift registers spanning the rows or columns. The increment and read-increment operators would increment the address without an additional decode. Due to time constraints, unexpectedly large layout areas, and uncertain power benefits, this approach was scrapped late in the design process. To avoid disturbing the rest of the design, the original memory interfaces were retained, but were implemented in a more conventional fashion, with incrementing registers storing the address decoded from with each access.

Because of the dual-rail nature of the 6T cell, the full-swing read operation is fully QDI. Operating full-swing limits the bank height (and therefore hurts density), but was easier to design because no sense-amplifiers were required.

Completion for the write operation is generated by computing data completion where it arrives at each column. This captures the propagation delay of the data to the bank, and pads it further with the time taken to negotiate the completion trees and return to the input. This intrinsic padding is augmented with a programmable delay line (which turned out to be unnecessary, in practice).

5.3.4.1 RW Memory

```
$RWMem(A, R, W) \equiv$
        $*[\overline{R} \longrightarrow R!\mathrm{M}[A?]$
          $[]\overline{W} \longrightarrow W?\mathrm{M}[A?]$
        $]$
```

CHP $\triangleright$ HSE

```
$RWMem(\mathbf{M}\mathrm{x}(1\mathrm{of}2)\, A,\ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\mathrm{of}2)}\ \mathrm{R}, \mathrm{W}) \equiv$
        $*[\overline{R.e} \longrightarrow [v(A)]; R.b := \mathrm{M}[A]; [\neg R.e \wedge n(A)]; R.b\downarrow$
          $[]\overline{v(W.b)} \longrightarrow [v(A)]; \mathrm{M}[A] := W.b; W.e\downarrow;$
            $[n(W.b) \wedge n(A)]; W.e\uparrow$
        $]$
```

5.3.4.2 RI Memory

RIMem is implemented by composing an RWMem, an IncReg, and some LpRa control processes.

```
$RIMem(A, RI, WI) \equiv$
        $*[\overline{A} \longrightarrow A?a$
          $[]\overline{RI} \longrightarrow RI!mem[a], a := a + 1$
          $[]\overline{WI} \longrightarrow WI?mem[a], a := a + 1$
        $]$
```

CHP ▷ HSE

```
$RIMem(\overleftrightarrow{\mathrm{e}\mathbf{M}\times(1\mathrm{of}2)}\ A,\ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\mathrm{of}2)}\ RI, WI) \equiv$
    $*[v(A.b) \longrightarrow a := A.b; A.e\downarrow; [n(A.b)]; A.e\uparrow$
      $[]RI.e \longrightarrow RI.b := mem[a]; [\neg RI.e], a := a+1; RI.b\downarrow$
      $[]v(WI.b) \longrightarrow mem[a] := WI.b; WI.e\downarrow; [n(WI.b)], a := a+1; WI.e\uparrow$
     ]
```

HSE ▷ HSE

```
$RI(\overleftrightarrow{\mathrm{e}\mathbf{M}\times(1\mathrm{of}2)}\ A,\ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\mathrm{of}2)}\ RI, WI) \equiv$


    $\mathrm{IncReg}(SET, GET, INC)\ inc\_reg$
    $\mathrm{RWmem}(aint, Rint, Wint)\ mem\_core$

    $*[v(A.b) \longrightarrow SET.b := A.b; [\neg SET.e]; A.e\downarrow;$
        $[n(A.b)]; SET.b\downarrow; [SET.e]; A.e\uparrow$
      $[]RI.e \longrightarrow Rint.e\uparrow, GET.e\uparrow; [v(GET.b)]; aint := GET.b;$
        $[v(Rint.b)]; RI.b := Rint.b;$
        $[\neg RI.e]; Rint.e\downarrow, GET.e\downarrow; [n(GET.b)]; aint\downarrow;$
        $[n(Rint.b)]; INC.d\uparrow;$
        $[\neg INC.e]; INC.d\downarrow;$
        $[INC.e]; RI.b\downarrow$
      $[]v(WI.b) \longrightarrow Wint.b := WI.b, GET.e\uparrow; [v(GET.b)]; aint := GET.b;$
        $[\neg Wint.e]; WI.e\downarrow;$
        $[n(WI.b)]; Wint.b\downarrow, GET.e\downarrow; [n(GET.b)]; aint\downarrow;$
        $[Wint.e]; INC.d\uparrow;$
        $[\neg INC.e]; INC.d\downarrow;$
        $[INC.e]; WI.e\uparrow$
     ]
```

A is wired directly to *SET*. The *RI* and *WI* branches are each sequenced using a *LpRa* process, letting the environment guarantee mutually exclusive access to the IncReg. We could have implemented more elaborate custom control to parallelize the *INC* operation with the subscriber's operations between reads and writes ($RI.b\downarrow$ and $WI.e\uparrow$ need not defer until $[INC.e]$).

5.3.4.3 RMW Memory

Since the user is responsible for the increment (it doesn't come automatically with the reads and writes), there is no internal control logic for the RMWmem:

```
$RMWmem(A, R, W, INC) \equiv$
    $*[\overline{A} \longrightarrow A?(a)$
      $[]\overline{R} \longrightarrow R!\mathrm{M}[a] \bullet W?\mathrm{M}[a];$
      $[]\overline{INC} \longrightarrow INC; a := a + 1$
    $]$
```

CHP ▷ HSE

```
$RMWmem(\overleftrightarrow{\mathrm{e}\mathbf{M}\times(1\mathrm{of}2)}\,A,\ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\mathrm{of}2)}\,R,\ \overleftrightarrow{\mathrm{e}\mathbf{N}\times(1\mathrm{of}2)}\,W,\ \overleftrightarrow{\mathrm{ed}}\,INC) \equiv$
    $*[v(A.d) \longrightarrow a := A.d; A.e\downarrow; [n(A.d)]; A.e\uparrow$
      $[]R.e \longrightarrow R.d := \mathrm{M}[a];$
          $[\neg R.e]; R.d\downarrow;$
          $[v(W.d)]; \mathrm{M}[a] := W.d; W.e\downarrow;$
          $[n(W.d)]; W.e\uparrow$
      $[]INC.d \longrightarrow a := a + 1, INC.e\downarrow; [\neg INC.d].INC.e\uparrow$
    $]$
```

HSE ▷ HSE

$RMWmem(\overleftrightarrow{\text{e}\mathbf{M}\times(\text{1of2})}\ A,\ \overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ R,\ \overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ DI,\ \overleftrightarrow{\text{ed}}\ INC) \equiv$

```
    IncReg$(SET, GET, INCint)$ $inc\_reg$
    RWmem$(aint, Rint, Wint)$ $mem\_core$

    $*[v(A.b) \longrightarrow SET.b := A.b;\ [\neg SET.e]\,; A.e\downarrow;$
        $[n(A.d)]\,; SET.b\downarrow;\ [SET.e]\,; A.e\uparrow$
      $[]R.e \longrightarrow Rint.e\uparrow, GET.e\uparrow;\ [v(GET.b)]\,; aint := GET.b;$
        $[v(Rint.b)]\,; RI.b := Rint.b;$
        $[\neg R.e]\,; Rint.e\downarrow, GET.e\downarrow;\ [n(GET.b)]\,; aint\downarrow;$
        $[n(Rint.b)]\,; R.b\downarrow$
      $[]v(W.b) \longrightarrow Wint.b := W.b, GET.e\uparrow;\ [v(GET.b)]\,; aint := GET.b;$
        $[\neg Wint.e]\,; W.e\downarrow;$
        $[n(W.b)]\,; Wint\downarrow, GET.e\downarrow;\ [n(GET.b)]\,; aint\downarrow;$
        $[Wint.e]\,; W.e\uparrow$
      $[]INC.d \longrightarrow INCint.d\uparrow;$
        $[\neg INCint.e]\,; INC.e\downarrow;$
        $[\neg INC.d]\,; INCint.d\downarrow;$
        $[INCint.e]\,; INC.e\uparrow$
     ]
```

5.3.5 Accumulator

Our ultimate goal in the Accumulator decomposition is to achieve some pipelining between of the AM and WM accesses: for each dimension, the next dimension's WM read can begin as the current dimension's AM data is being written back. This results in two semi-independent data and control paths, separated by buffering and synchronized by a central control process (Figure 5.2 on page 65), We begin by decomposing the top-level CHP program into several sub-programs, and make use of the RImem and RMWmem.

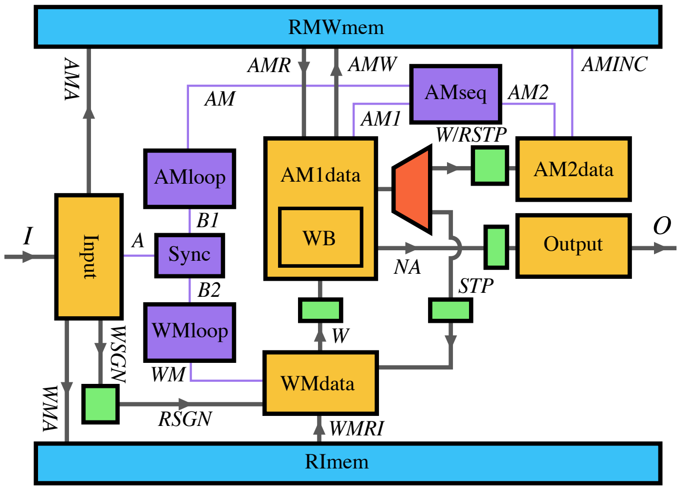


Figure 5.2: Accumulator subprocess composition. Control processes (purple blocks) trigger their associated data processes (yellow blocks) with dataless control channels (thin purple lines). PCFBs (green rectangles) allow for slack between AM and WM process halves. Registers (green squares) store the stop bit and sign of the input.

```
$Accumulator(I, O) \equiv$
        $stop := 0$
        $*[stop \longrightarrow$
          $I?(wma, ama, sign), stop := 0$
         $\Box \neg stop \longrightarrow$
          $d_{ij} := \mathrm{WM}[wma], (v, thr, stop) := \mathrm{AM}[ama];$
          $[sign = 1 \longrightarrow v' := v + d_{ij} \Box sign = -1 \longrightarrow v' := v - d_{ij}];$
          $trigger := v'[thr] \otimes \mathrm{sign}(v');$
          $[trigger \longrightarrow O!(tag\_out, \mathrm{sign}(v')), v'[thr] := \neg v'[thr]];$
          $\mathrm{AM}[ama].v := v, wma := wma + 1, ama := ama + 1$
         ]
```

CHP $\triangleright$ CHP

```
$WB(WBIO) \equiv$ port WBIO does an exchange communication
    $*[WBIO?(w,v,thr);$
      $[-thr < w+v < thr \longrightarrow$
        $v' := w+v, so := 0$
      $\square w+v > thr \longrightarrow$
        $v' := w+v-thr, so := +1$
      $\square w+v < -thr \longrightarrow$
        $v' := w+v+thr, so := -1$
      $];$
      $WBIO!(v', so)]$
```

```
$Accumulator(I,O) \equiv$

    WB$(WBIO)$ $wb\_datapath$
    RMWmem$(AMA, AMR, AMW, AMINC)$ $acc\_mem$
    RImem$(WMA, WMRI, WMWI)$ $weight\_mem$

    $*[stop \longrightarrow$
      $I?(ama, mma, sign); AMA!ama, WMA!mma$
     $\square \neg stop \longrightarrow$
      $WMRI?w, AMR?(stop, val, thr, tag\_out);$
      $WBIO!(w \times sign, val, thr); WBIO?(val', so);$
      $AMW!(stop, val', thr, tag\_out); AMINC;$
      $[so \neq 0 \longrightarrow O!(tag\_out, so) \square so = 0 \longrightarrow skip]$
     $]$
```

We perform control-data decomposition on the remaining process. We alter the control to allow for the pipelining of access to the two memories, which is advantageous because of the mismatch in access time between the large weight_mem and the smaller acc_mem (Figure 5.3 on page 67).

CHP $\triangleright$ CHP

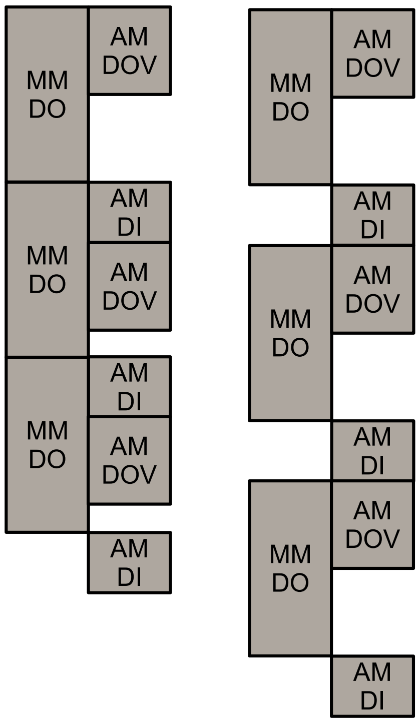


Figure 5.3: Pipelining the AM and WM halves of the datapath allows the sequencing on the left, instead of the sequencing on the right. The names are old: MM:*DO* = WM:*RI*, AM:*DI* and AM:*DOV* refer to the read and write parts of the AM:*RW* operation. By allowing the next WM lookup to begin before the AM writeback has completed, this results in an increase in throughput.

```
$Accumulator(I,O) \equiv$

    $\mathrm{WB}(WBIO)$ $wb\_datapath$
    $\mathrm{RMWmem}(AMA,AMR,AMW,AMINC)$ $acc\_mem$
    $\mathrm{RImem}(WMA,WMRI,WMWI)$ $weight\_mem$
    $\mathrm{Reg}(RSGN,WSGN)$ $sign\_reg$

    input process, sets addresses
    $*[A \bullet (I?(ama,wma,sign);AMA!ama,WMA!wma,WSGN!sign)]$

    synchronize input with AM and WM subprocesses
    $*[A \bullet B_1 \bullet B_2]$

    AM subprocess
    $*[stop \longrightarrow B_1 [] \neg stop \longrightarrow AM?stop]$
    $*[\overline{AM} \longrightarrow W?w,AMR?(stop,val,thr,na);STP!stop;$
        $WBIO!(w,val,thr);WBIO?(val',so);$
        $AMW!(stop,val',thr,na);AMINC;$
        $[so \neq 0 \longrightarrow O!(na,so) [] so = 0 \longrightarrow skip],$
        $AM!stop]$

    WM subprocess
    $*[stop \longrightarrow B_2 [] \neg stop \longrightarrow WM?stop]$
    $*[\overline{WM} \longrightarrow WMRI?w;RSGN?sign;W!(w \times sign);WM!STP?]$
```

We do further control-data decomposition for the AM subprocess, so that the resulting AM_1 and AM_2 operations can each be implemented by one four-phase handshake. The control process ensures that AMR/AMW will not overlap with $AMINC$.

$AM(AM, STP, O) \equiv$

```
    Reg$(RSTP, WSTP)$ $stop\_reg$

    $*[\overline{AM} \longrightarrow AM_1; AM_2?stop; AM!stop]$   control
    $*[AM_1 \bullet (W?w, AMR?(stop, val, thr, na);$   $AM_1$ data
      $STP!stop, WSTP!stop;$
      $WBIO!(w, val, thr); WBIO?(val', so);$
      $AMW!(stop, val', thr, na), NA!(na, so)]$
    $*[NA?(na, so);\ [so \neq 0 \longrightarrow O!(na, so) [] so = 0 \longrightarrow skip]]$
    $*[AM_2!RSTP? \bullet INC]$   $AM_2$ data
```

By putting some slack between the AM- and WM-loop branches of the program, we can achieve our pipelining. To allow the WM process to get ahead of the AM process, we insert a PCFB on STP and W. The NA process that came out of the above decomposition can also be pipelined if a PCFB is inserted on NA.

We now summarize the subprocesses we have generated. As noted, the control processes are equivalent to those that we have already described.

$$Input(A, I, AMA, WMA, WSGN) \equiv$$
$$[A \bullet (I?(ama, wma, sign); AMA!ama, WMA!wma, WSGN!sign)]$$

$$Sync(A, B_1, B_2) \equiv$$
$$*[A \bullet B_1 \bullet B_2]$$

$$AMloop(B_1, AM) \equiv WhileLoop(\cdot) \equiv$$
$$*[stop \longrightarrow B_1 [] \neg stop \longrightarrow AM?stop]$$

$$AMseq(AM, AM_1, AM_2) \equiv TarrowLaRaT(\cdot) \equiv$$
$$*[\overline{AM} \longrightarrow AM_1; AM_2?stop; AM!stop]$$

$$AM_1data(AM_1, W, AMR, STP, WSTP, WBIO, AMW, NA) \equiv$$
$$\begin{aligned}*[AM_1 \bullet (&W?w, AMR?(stop, val, thr, na); \quad AM_1 \;\; data\\ &STP!stop, WSTP!stop;\\ &WBIO!(w, val, thr); WBIO?(val', so);\\ &AMW!(stop, val', thr, na), NA!(na, so)]\end{aligned}$$

$$AM_2data(AM_2, RSTP, INC) \equiv$$
$$*[AM2!RSTP? \bullet INC] \quad AM_2 \;\; data$$

$$Output \equiv$$
$$*[NA?(na, so);\ [so \neq 0 \longrightarrow O!(na, so) [] so = 0 \longrightarrow skip]]$$

$$WMloop(B_2, WM) \equiv WhileLoop(\cdot) \equiv$$
$$*[stop \longrightarrow B_2 [] \neg stop \longrightarrow WM?stop]$$

$$WMdata(WM, WMRI, RSGN, W, STP) \equiv$$
$$*[\overline{WM} \longrightarrow WMRI?w; RSGN?sign; W!(w \times sign); WM!STP?]$$

Using the collected subprocesses, the memories, and standard circuitry, we compose the Accumulator (Figure 5.2 on page 65).

$Accumulator(I,O) \equiv$

RMWmem$(AMA, AMR, AMW, AMINC)$ *acc_mem*
RImem$(WMA, WMRI, WMWI)$ *weight_mem*

Input$(A, I, AMA, WMA, WSGN)$ *input*
Reg$(RGN, WSGN)$ *sign_reg*

Sync(A, B_1, B_2) *input_sync* *this is just wires and a C-element*

WhileLoop(B_1, AM) *AM_loop*
WhileLoop(B_2, WM) *WM_loop*

TarrowLaRaT(AM, AM_1, AM_2) *AM_seq*
AM$_1$data$(AM_1, Wbuf, AMR, STP, WSTP, WBIO, AMW, NA)$ *AM1_data*
WB$(WBIO)$ *writeback_datapath*
Reg$(RSTP, WSTP)$ *stop_reg*
AM$_2$data$(AM_2, RSTP, INC)$ *AM2_data*
PCFB$(STP, STPbuf)$ *STP_pcfb*
PCFB$(NA, NAbuf)$ *NA_pcfb*

Output$(NAbuf, O)$ *output*

WMdata$(WM, WMRI, RSGN, W, STPbuf)$ *WM_data*
PCFB$(W, Wbuf)$ *W_pcfb*

The jump to HSE is trivial for the datapath circuits, except for WB, the accumulator memory writeback datapath (See Section A.1 for more details).

5.3.6 Pool Action Table

The Pool Action Table is effectively just a RW memory and some wires.

```
$Pool\ \ Action\ \ Table(I,\ \ O) \equiv$
    $*[I?(subarray, neuron);$
      $(wma_{y,base}, wma_x, ama) := PAT\,[subarray]\,;$
      $O!((wma_{y,base}, neuron), wma_x, ama)]$
```

CHP $\triangleright$ CHP

```
$Pool\ \ Action\ \ Table(I,\ \ O) \equiv$

    RWmem$(R, W)$ $mem$

    $*[I?(subarray, neuron);$
      $R?(wma_{y,base}, wma_x, ama);$
      $O!((wma_{y,base}, neuron), wma_x, ama)]$
```

5.3.7 Tag Action Table

The Tag Action Table is relatively simple. The input tag is used to set the RImem's base address. In each loop execution, a sequential read is done from the memory. Some pipelining is afforded by breaking the processing of the read data into a second stage.

```
$TATStage1(I,S) \equiv$
        $stop := 0$
        $*[stop \longrightarrow I?(tag, ct)$
          $[]\neg stop \longrightarrow (stop, type, data) := TAT[tag];$
            $S!(type, data, ct), tag := tag + 1]$

$TATStage2(S, AO, SO, TO) \equiv$
        $*[S?(type, data, ct);$
          $[type = \textbf{GlobalTag} \longrightarrow$
            $(route, tag) := data; TO!(route, tag, ct)$
          $[]type = \textbf{SynapseSpike} \longrightarrow$
            $(sign_0, addr_0, sign_1, addr_1) := data;$
            $(SO!(sign_0 \otimes \mathrm{sign}(ct), addr_0),$
             $SO!(sign_1 \otimes \mathrm{sign}(ct), addr_1));$
            $TO!(0, tag, ct - \mathrm{sign}(ct))$
          $[]type = \textbf{AccumulatorInput} \longrightarrow$
            $(wma_x, wma_y, ama) := data;$
            $AO!(wma_x, wma_y, ama, \mathrm{sign}(ct)),$
            $TO!(0, tag, c - \mathrm{sign}(ct))]]$

$TAT(I, AO, SO, TO) \equiv$
        TATStage1$(I,S)$ $stage\_1$
        PCFB$(S, Sbuf)$ $S\_buf$
        TATStage2$(Sbuf, AO, SO, TO)$ $stage\_2$
```

We can decompose TATStage1 further to use the memory, and to use the WhileLoop for its control:

```
$INITdata(INIT, I, A) \equiv$
        $*[INIT \bullet (I?(tag, ct); A!tag)]$

$LOOPdata(LOOP, RI, S) \equiv$
        $*[LOOP \bullet (RI?(stop, type, data); S!(type, data, ct))]$

$TATStage1(I, S) \equiv$
        RImem$(A, RI, WI)$ *mem*
        WhileLoop$(INIT, LOOP)$ *control*
        INITdata$(INIT, I, A)$ *init_data*
        LOOPdata$(LOOP, RI, S)$ *loop_data*
```

TATStage2 has no control, but further decomposition can be performed to illustrate the reuse of the decrementer. Controlled splits and exclusive merges can be inferred from the final program.

```
$AbsDecrementer \equiv DECIO$
        $\text{Adder}(ADDIO)\ adder$
        $*[DECIO?ct; ADDIO!(ct, -\text{sign}(ct)); DECIO!ADDIO?]$

$TATStage2(S, AO, SO, TO) \equiv$
        $*[S?(type, data, ct);$
          $[type = \textbf{GlobalTag} \longrightarrow$
            $(route, tag) := data; TO!(route, tag, ct)$
          $[]type = \textbf{SynapseSpike} \longrightarrow$
            $(sign_0, addr_0, sign_1, addr_1) := data;$
            $(SO!(sign_0 \otimes \text{sign}(ct), addr_0),$
             $SO!(sign_1 \otimes \text{sign}(ct), addr_1));$
            $TO!(0, tag, DECIO!ct?)$
          $[]type = \textbf{AccumulatorInput} \longrightarrow$
            $(wma_x, wma_y, ama) := data;$
            $AO!(wma_x, wma_y, ama, \text{sign}(ct)),$
            $TO!(0, tag, DECIO!ct?)]]$
```

5.3.8 FIFO

We now present the decomposition of the most complicated single circuit in Braindrop, the FIFO. Aside from being a complicated process to begin with, the design was altered several times, and some of the complexity is residual in nature: sometimes there was pressure to shoehorn in existing circuits that weren't ideal for their newly assigned tasks. The CHP for the FIFO's two main pipeline stages is reproduced here (see Figure 5.1 on page 35 for the general structure of the ultimate process).

```
$DCTFIFO(I,P,G,OVFLW) \equiv$
        $*[\bar{I} \longrightarrow \;\; I?(tag, ct_{in});$
            $(d, ct_{curr}) := \mathrm{DCT}[tag];$
            $ct_{new} := ct_{curr} + ct_{in};$
            $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW]$
            $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW]$
            $\mathrm{DCT}[tag] := (1, ct_{new}), [\neg d \longrightarrow \;\; P!tag]$
          $|\overline{G} \longrightarrow \;\; G?tag; \mathrm{DCT}[tag] := (1, ct_{new}),$
            $(d, ct_{curr}) := \mathrm{DCT}[tag];$
            $\mathrm{DCT}[tag] := (0,0),$
            $[ct_{curr} \neq 0 \longrightarrow O!(tag, ct_{curr}])$
          $]$

$QFIFO(P,G) \equiv$
        $head := 0, tail := 0$
        $*[\overline{P} \longrightarrow \;\; P?tag, \mathrm{Q}[tail] := tag, tail := tail + 1$
          $|head \neq tail \longrightarrow G!\mathrm{Q}[head]; head := head + 1$
          $]$
```

5.3.8.1 DCTFIFO Pipeline Stage

We begin by decomposing the DCT stage. As previously mentioned, Braindrop separates traffic into two classes, occupying the upper and lower halves of the tag space. We duplicate each port for the 0 and 1 tag classes (the MSB of the tag ID) and instantiate the memory explicitly:

```
$DCTFIFO(I_0, I_1, P_0, P_1, G_0, G_1, OVFLW_0, OVFLW_1) \equiv$

    RWMem$(A, R, W)$ $DCT\_mem$

    $*[\overline{I_0} \longrightarrow$  $I_0?(tag, ct_{in})$;
        $A!(tag), R?(d, ct_{curr})$;
        $ct_{new} := ct_{curr} + ct_{in}$;
        $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW_0]$
        $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW_0]$
        $A!(tag), W!(1, ct_{new}), [\neg d \longrightarrow$  $P_0!tag]$
     $|\overline{G_0} \longrightarrow$  $G_0?tag$;
        $A!(tag), R?(d, ct_{curr})$;
        $A!(tag), W!(0, 0)$;
        $[ct_{curr} \neq 0 \longrightarrow O_0!(tag, ct_{curr})$
     $|\overline{I_1} \longrightarrow$  ... like $I_0$
     $|\overline{G_1} \longrightarrow$  ... like $G_0$
     ]
```

We want to make it possible to have the arithmetic of one branch of the arbitration occur in parallel with another branch's memory operation. We also want to avoid the possibility of blocking other branches because an output port is unable to accept a communication immediately. To achieve these objectives, we give each branch a parallel process, pushing the the arbitration towards the contended resource, the memory. The read/write operations of both input and output must appear atomic within a given tag class, so we cannot fully achieve our first goal: tag class 0's arithmetic may only occur in parallel with tag class 1's memory operations, and vice-versa.

```
$DCTin(I, A2I, RI, WI, P, OVFLW) \equiv$
        $instantiated\ \ once\ \ for\ \ each\ \ tag\ \ class$
        $*[I?(tag, ct);$
          $A2I!tag; RI?(d, ct);$
          $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW]$
          $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW]$
          $WI!(1, ct_{new});$
          $[\neg d \longrightarrow P!tag]]$
```

```
$DCTout(G, A2O, RO, WO, ZC) \equiv$
        $instantiated\ \ once\ \ for\ \ each\ \ tag\ \ class$
        $*[G?tag$
          $A2O!tag; RO?(d, ct); WO!(0,0),$
          $ZC!(ct, tag)]$
```

```
$DCTHalfArb(A2I, A20, A, RI, RO, R, WI, WO, W) \equiv$
        $implements\ \ atomic\ \ R/W\ \ within\ \ a\ \ class$
        $*[\overline{A2I} \longrightarrow A2I?a; A!a, RI!R?; A!a, W!WI?$
          $|\overline{A2O} \longrightarrow A2O?a; A!a, RO!R?; A!a, W!WO?$
         $]$
```

```
$MemArb(A_0, A_1, R_0, R_1, W_0, W_1) \equiv$
        $arbitrates\ \ memory\ \ access\ \ between\ \ classes$
        $*[\overline{A_0} \longrightarrow A_0?a$
            $[\overline{W_0} \longrightarrow W!W_0?$
            $[]\overline{R_0} \longrightarrow R_0!R?$
          $|\overline{A_1} \longrightarrow\ \ ...\ \ like\ \ A_0$
         $]$
```

```
$ZC(ZC, O) \equiv\ \ discards\ \ 0{\downarrow}count\ \ outputs$
        $*[ZC?(ct, tag); [\neg(ct = 0) \longrightarrow O!(tag, ct)]$
```

Using the above processes, we compose DCTMem:

$DCTFIFO(I_0, I_1, P_0, P_1, G_0, G_1, OVFLW_0, OVFLW_1) \equiv$

```
    RWMem$(A, R, W)$ $DCT\_mem$

    DCTin$(I_0, A2I_0, RI_0, WI_0, P_0, OVFLW_0)$ $DCT\_in\_0$
    DCTout$(G_0, A2O_0, RO_0, WO_0, ZC_0)$ $DCT\_out\_0$
    ZC$(ZC_0, O_0)$ $zero\_crusher\_0$
$DCTHalfArb \equiv A2I_0, A20_0, A_0, RI_0, RO_0, R_0, WI_0, WO_0, W_0 half\_arb\_0$

    DCTIn$(I_1, A2I_1, RI_1, WI_1, P_1, OVFLW_1)$ $DCT\_in\_1$
    DCTout$(G_1, A2O_1, RO_1, WO_1, ZC_1)$ $DCT\_out\_1$
    ZC$(ZC_1, O_1)$ $zero\_crusher\_1$
$DCTHalfArb \equiv A2I_1, A21_1, A_1, RI_1, RO_1, R_1, WI_1, WO_1, W_1 half\_arb\_1$
```

DCTin and DCTout are each implemented as two non-overlapping handshakes, to avoid tying up the memory if the output stage (either O_i or P_i) is unable to accept. DCTout's decomposition is:

$DCTin(I, A2I, RI, WI, P, OVFLW) \equiv$

```
    Reg$(Rreg, Wreg)$ $reg$
    LpRa$(L', R')$ $control$

    $*[L' \bullet (I?(tag, ct);$
      $A2I!tag; RI?(d, ct);$
      $[ct_{new} > ct_{+MAX} \longrightarrow ct_{new} := ct_{+MAX}, OVFLW]$
      $[ct_{new} < ct_{-MAX} \longrightarrow ct_{new} := ct_{-MAX}, OVFLW]$
      $WI!(1, ct_{new}),$
      $Wreg!(tag, d))$

    $*[R' \bullet (Rreg?(tag, d); [\neg d \longrightarrow P!tag]\,)]$
```

DCTHalfArb contains some custom control. It is decomposed as follows:

```
$A2(A2, A) \equiv$
    $\textit{repeats a single communication twice}$
    $*[A2?a \bullet (A!(a, rd); A!(a, wr))]$

$DCTHalfArbRMerge(A2I, A20, A2, RI, RO, R) \equiv$
    $\textit{does I/O arbitration, remembers choice to direct R to RI or RO}$
    $*[[\overline{A2I} \longrightarrow A2I?a; A2!a; RI!R?$
      $|\overline{A2O} \longrightarrow A2O?a; A2!a; RI!R?$
     $]]$

$DCTHalfArb(A2I, A20, A, RI, RO, R, WI, WO, W) \equiv$
    $\text{A2}(A2, A)\ \textit{address\_repeater}$
    $\text{DCTHalfArbRMerge}(A2I, A2O, A2, RI, RO, R)\ \textit{arb\_and\_R\_merge}$
    $\text{ExclusiveMerge}(WI, WO, W)\ \textit{W\_merge}$
```

A2 uses a new control circuit with the following CHP and HSE. This isn't quite control-data decomposition, so full HSE is presented.

```
$A2(\overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ L, R,\ \ \text{bool}\ rd, wr) \equiv$
    $*[[v(L.b)]; R.b := L.b, rd\uparrow;$
      $[\neg R.e]; R.b\downarrow, rd\downarrow;$
      $[R.e]; R.b := L.b, wr\uparrow;$
      $[\neg R.e]; L.e\downarrow;$
      $[n(L.b)]; R.b\downarrow, wr\downarrow;$
      $[R.e]; L.e\uparrow]$
```

HSE ▷ HSE + PRS

```
$A2control(\overleftrightarrow{\text{ed}}\ L,R,\ \text{bool}\ rd,wr) \equiv$
        $*[[L.d];rd\uparrow;$
            $[\neg R.e];R.d\downarrow,rd\downarrow;$
            $[R.e];a\downarrow;R.d\uparrow,wr\uparrow;$
            $[\neg R.e];L.e\downarrow;$
            $([\neg L.d];RWe\downarrow),a\uparrow;$
            $[R.e];L.e\uparrow]$
```

```
$A2data(\mathbf{N}\text{x}(1\text{of}2)\ L,R,\ \text{bool Rd} \equiv$
        $\text{AND}(L[:].t,Rd,R[:].t)\ L\_to\_R\_t[\mathbf{N}]$
        $\text{AND}(L[:].f,Rd,R[:].f)\ L\_to\_R\_f[\mathbf{N}]$
```

```
$A2(\overleftrightarrow{\text{e}\mathbf{N}\times(1\text{of}2)}\ L,R,\ \text{bool}\ rd,wr) \equiv$
        $\overleftrightarrow{\text{ed}}\ L',R'$ control versions of channels
        $\text{OR}(L.b[0].f,L.b[0].t,L'.d)$
        $L'.e = L.e$
        $\text{bool}\ Rd = R'.d$
        $R'.e = R.e$
        $\text{A2control}(L',R',wr,rd)\ control$
        $\text{A2data}(L.b,R.b,R'.d)\ data$
```

A2control is actually implemented by two independent processes, one which controls the sequencing of the outputs with respect to the input, and another which generates alternating *rd* and *wr* commands:

```
$A2seq(\overleftrightarrow{\text{ed}}\ L,R,\ \text{bool}\ RWe) \equiv$
        $*[[L.d];RWe\uparrow;$
          $[\neg R.e];R.d\downarrow,RWe\downarrow;$
          $[R.e];a\downarrow;R.d\uparrow,RWe\uparrow;$
          $[\neg R.e];L.e\downarrow;$
          $([\neg L.d];RWe\downarrow),a\uparrow;$
          $[R.e];L.e\uparrow]$

$A2rw(\overleftrightarrow{\text{e(1of2)}}\ RW) \equiv$
        implemented as a modifed $*[L;R]$ process
        $*[[RW.e];RW.rd\uparrow;[\neg RW.e];RW.rd\downarrow;$
          $[RW.e];RW.wr\uparrow;[\neg RW.e];RW.wr\downarrow]$

$A2control(\overleftrightarrow{\text{ed}}\ L,R,\ \text{bool}\ rd,wr) \equiv$
        A2seq$(L,R,RW.e)$ $seq\_ctrl$
        A2rw$(RW)$ $rw\_ctrl$
        $rd = RW.rd$
        $wr = RW.wr$
```

HSE ▷ PRS

$A2seq(\overleftrightarrow{\text{ed}}\ L,R,\ \text{bool}\ RWe) \equiv$

```
    $\neg\_a \ \wedge \ \neg\_Re \ \longrightarrow \ Le\uparrow$
    $\_a \ \wedge \ \_Re \ \longrightarrow \ Le\downarrow$

    $\neg\_Le \ \wedge \ \neg R.d \ \wedge \ \neg\_Re \ \longrightarrow \ \_a\uparrow$
    $pReset \ \vee \ \_Le \ \wedge \ R.d \ \longrightarrow \ \_a\downarrow$

    $\_sReset \ \wedge \ \_\_a \ \wedge \ Le \ \wedge \ \_Re \ \longrightarrow \ R.d\downarrow$
    $\neg\_sReset \ \vee \ \neg\_\_a \ \wedge \ \neg\_Re \ \longrightarrow \ R.d\uparrow$

    $\_a \ \Longrightarrow \ \_\_a\downarrow$
    $R.e \ \Longrightarrow \ \_Re\downarrow$
    $Le \ \Longrightarrow \ \_Le\downarrow$

    $L.d \ \wedge \ R.d \ \Longrightarrow \ \_RWe\downarrow$
    $\_RWe \ \Longrightarrow \ RWe\downarrow$
```

5.3.8.2 QFIFO Pipeline Stage

We now return to the QFIFO stage, rewriting it with ports for each tag class and instantiating the memory explicitly

$QFIFO(P_0, P_1, G_0, G_1) \equiv$

```
    $\text{RWmem}(R, W, A)\ Q\_mem$

    $head := 0, tail := 0$
    $*[\overline{P_0} \longrightarrow P_0?tag, A!tail_0, W!tag; tail_0 := tail_0 + 1$
      $| head_0 \neq tail_0 \longrightarrow A!head_0, G_0!R?; head_0 := head_0 + 1$
      $| \overline{P_1} \longrightarrow \ ...\ like\ \overline{P_0}$
      $| head_1 \neq tail_1 \longrightarrow \ ...\ like\ head_0 \neq tail_0$
     ]
```

We would like to do what we did for DCT, breaking this into four processes and moving the arbitration closer to the memory. This is easier than before because each branch only does a single memory access, (a read or a write instead of both), but harder than before because the branches in each class have to share information about the $head_i \neq tail_i$ condition. Consequently, we need to arbitrate the access to $head_i$ and $tail_i$ as well. We re-compile using the resulting HT process as follows:

```
$HT(T, H, TREL, HREL, INIT) \equiv$
        $*[\,[\overline{T} \longrightarrow T!tail; tail := tail+1, empty\downarrow; TREL$
            $|\,\overline{H} \wedge \neg empty \longrightarrow; H!head; head := head+1, empty := head = tail; HREL$
          $[]INIT \longrightarrow tail := 0, head := 0$
         $]]$

$QFIFO(P_0, P_1, G_0, G_1) \equiv$

        RWmem$(R, W, A)$ $Q\_mem$
        HT$(H_0, T_0, TREL_0, HREL_0, INIT_0)$ $HT\_0$
        HT$(H_1, T_1, TREL_1, HREL_1, INIT_1)$ $HT\_1$

        $*[\overline{P_0} \longrightarrow P_0?tag, A!T_0?, W!tag; TREL_0$
           $|\,head_0 \neq tail_0 \longrightarrow A!H_0?, G_0!R?; HREL_0$
           $|\,\overline{P_1} \longrightarrow$ ... $like\ \overline{P_0}$
           $|\,head_1 \neq tail_1 \longrightarrow$ ... $like\ head_0 \neq tail_0$
         $]$
```

The operations in the P_i and $head_i \neq tail_i$ branches may be compiled much in the same was as the DCTFIFO's branches were, with a similar memory arbitration process and $*[L;R]$ control processes for each branch.

The HT process, however, requires custom decomposition:

```
$HTreg(TR, HR, INIT) \equiv$

      Reg$(HRR, HRW)$ $head\_reg$
      Reg$(TRR, TRW)$ $tail\_reg$
      Reg$(SR, SW)$ $slack\_reg$
      $*[[\overline{TR} \longrightarrow TRR?t; INCI!t?t'; SW!t', TR!t;$
            $TRW!SR?$
        $[]\overline{HR} \longrightarrow HRR?h; TRR?t; INCI!h?h';$
            $CMP!(h', t)?e; SW!h', HR!(h, e);$
            $HRW!SR?$
       ]]

      $*[CMP?(x, y)!(x = y)]$
      $*[INCI?x; x' := (x+1)\%MAX; INCI!x']$

$HTarb(T, H, TREL, HREL, TR, HR) \equiv$

      $*[\overline{T} \longrightarrow T!TR?, empty\downarrow; TREL$
        $|\overline{H} \wedge \neg empty \longrightarrow HR?(h, empty); H!h; HREL$
       ]

$HT(T, H, TREL, HREL, INIT) \equiv$

      HTreg$(TR, HR, INIT)$ $HT\_reg$
      HTarb$(T, H, TREL, HREL, TR, HR, INIT)$ $HT\_arb$
```

HREL and TREL signal the release of the arbitration hold. Doing control-data decomposition on HTreg, we get the following CHP programs, which we decompose into HSE and PRS:

$HTregcontrol(TA,TB,TC,HA,HB,HC) \equiv$

```
    $*[\overline{TA} \longrightarrow TA,TB;TC$
      $[]\overline{HA} \longrightarrow HA,HB;HC$
     $]$
```

$TATBdata(TA,TB,TRR,INCI,SW) \equiv$

these overlapping bullets aren't very explanatory, see the HSE below

$$*[TA \bullet TB \bullet (TRR?t;INCI!t?t';SW!t',TR!t)]$$

$TCdata(TC,TRW,SR) \equiv$

$$*[TC \bullet (TRW!SR?)]$$

$HAHBdata(HA,HB,HRR,TRR,INCI,CMP,SW,HR) \equiv$

$$*[HA \bullet HB \bullet (HRR?h;TRR?t;INCI!h?h';CMP!(h',t)?e;SW!h',HR!(h,e))]$$

$HCdata(HC,HRW,SR \equiv$

$$*[HC \bullet (HRW!SR?)]$$

CHP ▷ HSE

```
$HTregcontrol(\overleftrightarrow{\text{ed}}\ TA,TB,TC,HA,HB,HC) \equiv$
    $*[[TA.e \longrightarrow TA.d\uparrow, TB.d\uparrow;$
        $[\neg TA.e \wedge \neg TB.e];a\downarrow;TA.d\downarrow,TB.d\downarrow;$
        $[TB.e];TC.d\uparrow;$
        $[\neg TC.e];a\uparrow;TC.d\downarrow$
        $[TC.e]$
      $[]HA.e \longrightarrow$
        $\ldots$ same as $TA.e$
     $]]$

$TATBdata(\overleftrightarrow{\text{ed}}\ A,B,\ \overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ TR,INCI,SW) \equiv$
    $*[[TR.e];A.e\uparrow;$
      $[A.d \wedge B.d];TRR.e\uparrow;$
      $[v(TRR.d)];TR.d := TRR.d, INCI.d := TRR.d;$
      $[v(INCO.d)];SW.d := INCO.d;$
      $([\neg TR.e];A.e\downarrow),([\neg SW.e];B.e\downarrow);$
      $[\neg A.d \wedge \neg B.d];TRR.e\downarrow;$
      $[n(TRR.d)];TR.d\downarrow, INCI.d\downarrow;$
      $[n(INCO.d)];SW.d\downarrow;$
      $[SW.e];B.e\uparrow]$

    HAHBdata is quite similar to TATBdata, but with more complicated operators
```

CHP ▷ PRS

$HTregcontrol(\overleftrightarrow{\text{ed}}\ TA,TB,TC,HA,HB,HC) \equiv$

$$a \ \wedge \ b \ \wedge \ TC.e \ \wedge \ HC.e \ \wedge \ \_\_TAe \ \longrightarrow \ \_TAd\downarrow$$
$$\neg\_pReset \ \vee \ \neg a \ \wedge \ \neg\_\_TAe \ \longrightarrow \ \_TAd\uparrow$$

$$\neg a \ \wedge \ \neg\_TBe \ \longrightarrow \ TC.d\uparrow$$
$$a \ \vee \ \_TBe \ \longrightarrow \ TC.d\downarrow$$

$$\_TAe \ \wedge \ \_TBe \ \longrightarrow \ a\downarrow$$
$$\neg\_pReset \ \vee \ \neg TC.e \ \longrightarrow \ a\uparrow$$

$$TB.e \ \Longrightarrow \ \_TBe\downarrow$$
$$TA.e \ \Longrightarrow \ \_TAe\downarrow$$
$$\_TAe \ \Longrightarrow \ \_\_TAe\downarrow$$

$$\neg\_TAd \ \longrightarrow \ TA.d\uparrow$$
$$\_TAd \ \longrightarrow \ TA.d\downarrow$$

the HA/HB/HC side is symmetric to the above

HTarb compiles as follows:

$HTarbcontrol(T,H,TREL,HREL) \equiv$

```
*[$\overline{T} \longrightarrow T, empty\downarrow; TREL$
  |$\overline{H} \wedge \neg empty \longrightarrow H?empty; HREL$
 ]
```

This is compiled using a standard arbiter:

```
$HTarbcontrol(\overleftrightarrow{\mathrm{ed}}\ T,H,TREL,HREL,\ \ \mathrm{1of2}\ htempty) \equiv$

    $htempty\ \ comes\ \ on\ \ with\ \ H.d\uparrow$

    $\mathrm{Arbiter}(a,b,u,v)\ arb$

    $*[[T.e];a\uparrow;$
      $[u];T.d\uparrow,empty\downarrow;$
      $[\neg T.e];T.d\downarrow;$
      $[TREL.d];a\downarrow;TREL.e\downarrow;$
      $[\neg u \wedge \neg TREL.d];TREL.e\uparrow]$

    $*[[n(H.e) \wedge \neg empty];b\uparrow;$
      $[v];H.d\uparrow;$
      $[htempty.t \longrightarrow empty\uparrow [] htempty.f \longrightarrow skip];$
      $[\neg H.e];H.d\downarrow;$
      $[n(htempth) \wedge HREL.d];b\downarrow;HREL.e\downarrow;$
      $[\neg v \wedge \neg HREL.d];HREL.e\uparrow]$

HSE $\triangleright$ PRS
```

$HTarb(\overleftrightarrow{\text{e}\mathbf{N}\times(\text{1of2})}\ T, H, TR, HR,\ \overleftrightarrow{\text{ed}}\ TREL, HREL) \equiv$

Arbiter(a, b, u, v) *arb*

T branch

$T.e \ \wedge\ TREL.e \ \longrightarrow\ \_a\downarrow$

$\neg\_pReset \ \vee\ \neg T.e \ \wedge\ \neg TREL.e \ \longrightarrow\ \_a\uparrow$

$T.e \ \wedge\ u \ \longrightarrow\ \_Td\downarrow$
$\neg T.e \ \vee\ \neg u \ \longrightarrow\ \_Td\uparrow$

$u \ \wedge\ TREL.d \ \longrightarrow\ TREL.e\downarrow$
$\neg u \ \wedge\ \neg TREL.d \ \longrightarrow\ TREL.e\uparrow$

H branch

$\_sReset \ \wedge \ H.e \ \wedge \ HREL.e \ \wedge \ \_empty \ \longrightarrow \ \_b\downarrow$
$\neg\_sReset \ \vee \ \neg H.e \ \wedge \ \neg HREL.e \ \longrightarrow \ \_b\uparrow$

$\neg\_pReset \ \vee \ \neg H.e \ \wedge \ (\neg\_HTempty.f \ \vee \ \neg\_empty) \ \longrightarrow \ \_Hd\uparrow$
$H.e \ \wedge \ v \ \longrightarrow \ \_Hd\downarrow$

$v \ \wedge \ HREL.d \ \wedge \ \_HTempty.t \ \wedge \ \_HTempty.f \ \longrightarrow \ HREL.e\downarrow$
$\neg v \ \wedge \ \neg HREL.d \ \longrightarrow \ HREL.e\uparrow$

$u \ \longrightarrow \ empty\downarrow$
$\neg\_pReset \ \vee \ \neg\_HTempty.t \ \longrightarrow \ empty\uparrow$

$empty \ \Longrightarrow \ \_empty\downarrow$
$HTempty.t \ \Longrightarrow \ \_HTempty.t\downarrow$
$HTempty.f \ \Longrightarrow \ \_HTempty.f\downarrow$
$\_Td \ \Longrightarrow \ T.d\downarrow$
$\_Hd \ \Longrightarrow \ H.d\downarrow$
$\_a \ \Longrightarrow \ a\downarrow$
$\_b \ \Longrightarrow \ b\downarrow$

Chapter 6

Evaluation and Future Work

6.1 Power and Throughput Measurements by Component

We measured the energy per operation for several of Braindrop's digital components (Table 6.1). The measurements seem consistent with each other. FIFO power is the highest because each traversal of the FIFO involves 6 memory operations (read + write for the input and output DCT operations, and read or write for the PG operations). The TAT+AER:RX operation involves a single memory read, and is somewhat less expensive. Accumulator power is about double the TAT power, in spite of involving a much larger memory and adding the PAT and accumulator read/write operations.

	Energy/op (pJ)	Throughput (MHz)
AER:RX	X	18.3
AER:TX	X	27.0
PAT + Accumulator (E_{d})	15.12	65.6
FIFO (E_{f})	28.27	X
TAT + AER:RX (E_{e})	7.55	X
E_{op} ($N/D = 64$, $R_{\mathrm{g}} = 20$)	.388	N/A

Table 6.1: Component throughput and energy/operation. Measurements marked X were not feasible.

Active and static power for the analog components is negligible compared to either component of digital power. We wanted to also report digital static leakage, which should dominate overall system power, but because of a foundry issue, it is an order of magnitude higher than expected. We expected to be dominated by SRAM leakage, which should have been about 20 μA for all of the memories.

6.2 Comparison to Other Architectures

Without a clear set of benchmarks, the efficiency of neuromorphic architectures is typically measured in energy per synaptic operation (Merolla et al., 2014; Davies et al., 2018), but even what constitutes a synaptic operation is ill-defined. The value of this quantity depends on how weight matrices are implemented (e.g. sparse, low-rank representations), whether network size necessitates inter-core communication, and if different signal representations are used, it should also account for their precision. For Braindrop, the most important parameter is the rank of the encode and decode matrices.

To report our efficiency, we compute Braindrop's energy per *equivalent synaptic operation*, counted using the throughput of a network using a deterministically-weighted, $N \times N$, dense matrix that achieves the same SNR at each synapse. This shows how our efficiency improves as the rank of the synaptic weight matrix being implemented decreases, and how it varies with the desired SNR, compared to the reference fully-connected approach.

First, we compute the power used to implement a $N-D-N$ decode-encode on Braindrop, for a given SNR observed by the synapses, which implies a certain throughput. For standard-basis anchor encoders, each synapse receives the output of a single accumulator. Equation 1 may be inverted to obtain $R_\text{p}(R_\text{g}, k)$, the SNR of the Poisson process, that when thinned by k, produces accumulator output with SNR R_g. Since $R_\text{p} = \sqrt{2F_\text{in}}$ (with F_in in units $1/\tau$), each neuron therefore spikes at $F_\text{spk} = F_\text{in}/N$. Total throughput for the decode is therefore $T_\text{d} = NDF_\text{spk} = DF_\text{in} = DR_\text{p}^2/2$. The D accumulators emit thin T_d by k, offering throughput $T_\text{f} = DR_\text{p}^2/(2k)$ to the FIFO. Each stream fans out to P tap points, giving total sparse encode throughput $T_\text{e} = DPR_\text{p}^2/(2k)$. For decode, FIFO, and sparse encode energies per operation E_d, E_f and E_e, the total power consumed is therefore

$$\begin{aligned} P_{BD} &= E_\mathrm{d}T_\mathrm{d} + E_\mathrm{f}T_\mathrm{f} + E_\mathrm{e}T_\mathrm{e} \\ &= D\frac{R_\mathrm{p}(R_\mathrm{g},k)^2}{2}\left(E_\mathrm{d} + \frac{1}{k}E_\mathrm{f} + \frac{P}{k}E_\mathrm{e}\right) \end{aligned}$$

$$\text{where} \quad R_\mathrm{p}^2(R_\mathrm{g},k) = \frac{1}{2}R_\mathrm{g}^2\left(1 + \sqrt{1 + \frac{4}{3}\frac{k^2}{R_\mathrm{g}^2}}\right)$$

We now derive the equivalent number of synaptic operations per second. We compare ourselves to a fully-connected network with N neurons (implemented deterministically with weights of equal value) that achieves an SNR of $R_\mathrm{FC} = R_g$ at each synapse. Each synapse receives a Poisson input whose rate is equal to the sum of all the neuron's rates. Using the same equations as before, $F_\mathrm{spk} = F_\mathrm{in}/N = R_\mathrm{FC}^2/(2N) = R_\mathrm{g}^2/(2N)$, and $T_{FC} = N^2F_\mathrm{spk} = NR_\mathrm{g}^2/2$.

To obtain energy per equivalent synaptic operation, we now divide the power consumed by Braindrop's network by the number of synaptic operations per second in the equivalent fully-connected network. The number of tap points per dimension is given by $P = \rho N/D$, where ρ is the density of tap points (i.e. tap points per neuron). The resulting expression shows how Braindrop's efficiency scales with N/D and with the desired SNR, R_g:

$$\begin{aligned} E_{\widetilde{\mathrm{op}}} &= \frac{P_\mathrm{BD}}{T_{FC}} \\ &= \frac{D}{N}\frac{R_\mathrm{p}(R_\mathrm{g},k)^2}{R_\mathrm{g}^2}\left(E_\mathrm{d} + \frac{1}{k}E_\mathrm{f} + \frac{\rho N}{kD}E_\mathrm{e}\right) \\ &= \frac{R_\mathrm{p}(R_\mathrm{g},k)^2}{R_\mathrm{g}^2}\left(\frac{D}{N}E_\mathrm{d} + \frac{D}{Nk}E_\mathrm{f} + \frac{\rho}{k}E_\mathrm{e}\right) \\ &= \frac{1}{2}\left(1 + \sqrt{1 + \frac{4}{3}\frac{k^2}{R_g^2}}\right)\left[\frac{D}{N}\left(E_\mathrm{d} + E_\mathrm{f}\frac{1}{k}\right) + E_\mathrm{e}\frac{\rho}{k}\right] \end{aligned}$$

k is a free parameter which we can optimize for at each SNR. Using our measured values for E_d, E_f, and E_e (Tab. 6.1) and $\rho = 8$ (see Sec. 3.2), for a desired synaptic SNR of 20 and 64 neurons per dimension (a typical operating point for NEF) the energy per synaptic

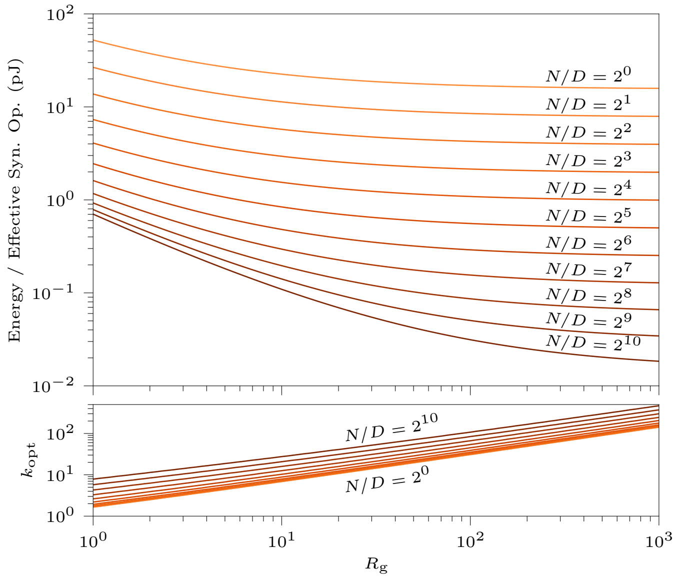


Figure 6.1: Energy per equivalent synaptic operation on Braindrop for varying ratios of N to D (top), for the optimal value of k (bottom). For each synaptic SNR R_g, total power is computed decode-encode networks achieving that synaptic SNR. This is divided by the throughput required by a fully-connected network achieving the same synaptic SNR. Braindrop is more efficient when implementing lower rank matrices and relatively high synaptic SNRs. In comparison, TrueNorth consumes 21 pJ/op for typical network configurations, and Loihi consumes a minimum of 24 pJ/op (Merolla et al., 2014; Davies et al., 2018).

operation is 388 fJ. As expected, Braindrop excels when implementing matrices with low rank relative to the number of neurons, and is at an even higher advantage compared to the fully-connected network when higher SNRs are required (Figure 6.1 on page 95).

6.3 NEF Benchmark Performance

Running NEF benchmarks networks allow us to demonstrate that our efforts to support NEF's programming abstractions have succeeded. Unfortunately, the degree of mismatch on the fabricated chip is several times what we were led to believe from the foundry's device models, so the number of neurons needed to implement NEF populations must be increased substantially over reference NEF implementations.

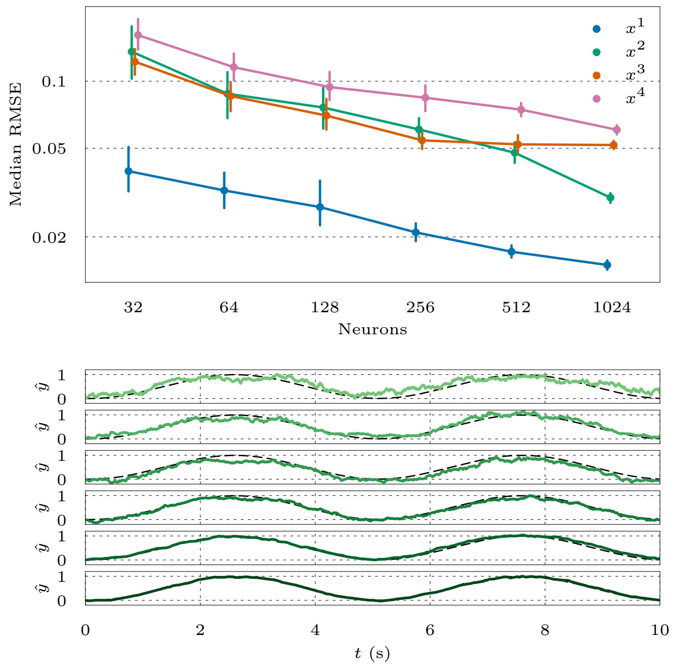


Figure 6.2: Decode performance for increasingly difficult polynomial functions on Braindrop. *Top*: Performance decoding x^D, $D \in [1,4]$ is reported. At each D, different pool sizes are used. For each configuration, the experiment is repeated 20 times, with 5-95% confidence intervals shown around the median RMSE. *Bottom*: sample outputs of populations decoding $\hat{y} \approx x^2$ (green), plotted against the ideal output, y, (dashed black) for sinusoidally varying input x. Each sub-panel corresponds to a particular number of neurons, increasing from 32 to 1024 by powers of two from top to bottom.

The first test of Braindrop's performance was to map 1D functions, sweeping the number of neurons and function difficulty (polynomial order) and measuring the RMSE of the approximation (Figure 6.2 on page 96). The computation was described in Nengo and mapped to the hardware. As expected, having fewer neurons, or having to decode a harder function, degrades performance.

It is also important to ensure that Braindrop can implement NEF networks that use dynamics, to ensure that the synapse is operating properly. An integrator, $\dot{x} = I$, is the most basic dynamical system that we can implement (the identity function, $y = x$ is decoded, scaled, and fed back to the inputs). Since unexpected behavior can arise from errors in

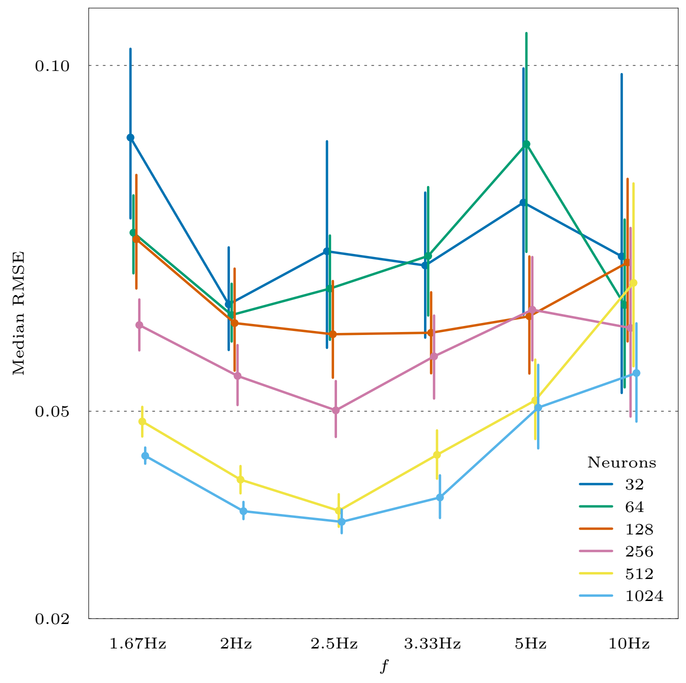


Figure 6.3: Integrator performance on an input of $\cos(2f\pi x)$, for varying f and pool size. There was an unexplained shift in the output waveforms while running this trial, so the errors reported are for shifted versions of the output (with the amount of shift that minimizes the error)

approximation of identity, more neurons is helpful in this setting as well (Figure 6.3 on page 97).

To validate our use of tap points to implement encoders, it is also important to show that we can decode functions of multiple dimensions. To demonstrate this, we decode a 2D vector rotation:

$$y = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} x \tag{6.1}$$

To implement this slightly more easily, we compute $\cos\theta$ and $\sin\theta$ in the computer and feed them in. To perform the matrix-vector multiply, we factorize the computation into four pools implementing multiplications $(z = x \cdot y)$ and combine their outputs. This is therefore foremost a test of our ability to implement simple multiplication.

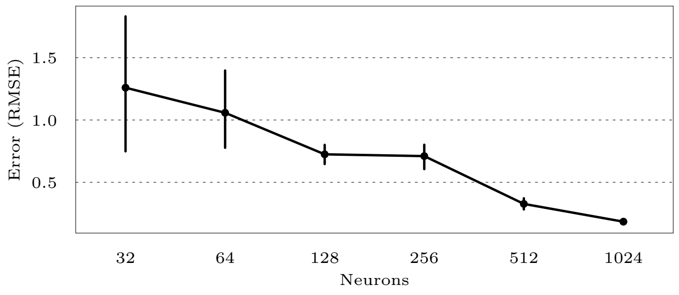


Figure 6.4: Performance of 2D vector rotation for increasing numbers of neurons.

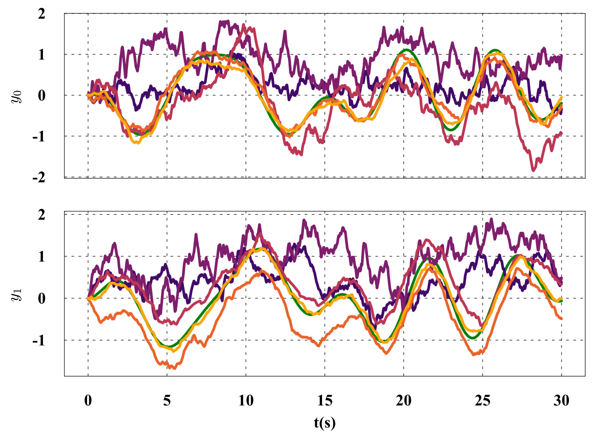


Figure 6.5: Sample traces of 2D vector rotation outputs. As the number of neurons is increased from 32 to 1024 in powers of two (purple to yellow lines), the output trace converges to the ideal output (green).

As would be expected, more neurons aids the approximation (Figure 6.4 on page 98). Examining the traces shows that performance is initially terrible, but improves gradually until the approximation is reasonably good (Figure 6.5 on page 98).

6.4 Future Work

Future work should continue on two fronts: optimizing Braindrop's physical design and exploring new computational frameworks that use spikes in more compelling ways than the NEF does. Braindrop's physical design was rushed in some areas, and too much effort was put into other areas that ultimately had only a small effect on overall system behavior. The NEF provided us with a computational vehicle which was well-suited to our hardware, and has been demonstrated as being scalable to very large systems. However, its efficiency is limited by the neurons' lack of coordination in their spiking: the signal coming from the neurons is effectively Poisson-encoded.

Braindrop's power consumption is dominated by static leakage, and future work should go towards minimizing this as much as possible. A conservative design-time estimate for the static leakage was 100 μA. Even this is considerable when compared to the simplest microcontrollers (which, at least for simple NEF applications, provide perhaps the best basis of comparison), which burn tens of μAs per MHz (and much less when put into a low-power state, useful if computations can be performed infrequently). This suggests that Braindrop is probably best applied to applications that must be performed continuously, such as those that have intrinsic dynamics.

If SRAM is to be used in the future, work should be done to optimize the bitcell for leakage power, even if it comes at the expense of density. A more appealing (but less-obviously immediately adoptable) alternative might be to leverage non-volatile memory (NVM) technologies. We have some flexibility in this regard, since SRAM's high access speed is not really needed in our application. SRAM might be retained where write operations must be performed during operation, such as for the accumulator memory and FIFO. Elimination of array leakage power is likely to simply push the problem to periphery power, so if NVM was to be adopted, it would probably be worth considering moving to a less pipelined architecture that uses fewer memories (as it stands, the speed offered by the pipelining is unnecessary).

SRAM density could be improved by more effort in designing custom logic for the SRAM peripheries, which are currently mostly standard-cell based. The granularity of the hierarchical wordline should also probably be increased for the weight memory (because of the small word size, almost 40% of the memory core's area is actually taken by the sub-wordline driver circuits, and the dummy cells used to isolate them from the bitcells). This would save dynamic power for 2D+ decodes, but use more for 1D decodes.

The FIFO is a particularly complicated datapath circuit, and should probably be simplified in future designs. The easiest thing to do would be to give each tag class a dedicated

FIFO. This would come at the expense of some additional SRAM periphery area overhead, but would be recouped to some extent by removing the structures needed to arbitrate between the two parallel FIFO datapaths.

Braindrop's standard cells could have been substantially more dense. We initially attempted to lay out the standard cells automatically, selecting a cell height that it seemed like the tool could handle. In the end, however, we ultimately re-did most of the layouts by hand. In retrospect, we should have used the less-aggressive of the two foundry library cell heights. This would have at least let us use the foundry cells for combinational logic.

Braindrop's architecture attempts to get the most out of every spike that the neurons produce, but another approach is to move from NEF to a computational framework that simply produces fewer spikes in the first place. One promising approach is described in Boerlin et al. (2013). At the population level, this approach still implements a rate code, but with a periodic spike coding. The key feature of this approach is the fast recurrent inhibition that ensures that after a neuron spikes, neurons with similar encoding vectors will be prevented from firing immediately. The neurons ultimately take turns emitting spikes, leading to the periodic output. This framework is only applicable to linear dynamical systems, and cannot be used for nonlinear computations, however. It is fully compatible with the NEF, however: it is possible to imagine a system that combined the two.

6.5 Concluding Remarks

Braindrop unites analog efficiency and digital programmability, providing an NEF-based synthesis process mapping high-level abstractions to spiking subthreshold analog neurons. Realizing analog circuits' efficiency was only possible through optimizing NEF operators to minimize digital communication without violating the abstractions they present to the user. Braindrop presents two such innovations: tap-points and the accumulator, which together allow for a massive reduction in digital traffic while remaining nominally invisible to the user. The hardware modules implementing these operators also had to be organized with transparency in mind, to minimize the possibility that mapping could be constrained by physical restrictions arising from resource allocation. In exchange for some small up-front area costs, we were able to ensure high utilization of the area-dominant weight memory and neurons. The application results demonstrate the synthesis process running on the hardware, realizing the goals of the project. Braindrop required co-design of all layers of the system architecture, keeping a theoretical framework in mind even at the lowest levels of the hardware design. This painstaking process has resulted in a new computational platform with hardware that embodies the brain's microarchitectural techniques that runs behind an accessible programming framework.

Appendix A

Datapath Decomposition Details

A.1 WB Datapath Implementation

The WB process uses one's complement addition to more easily detect and correct the over/under-threshold conditions (Figure A.1 on page 103).

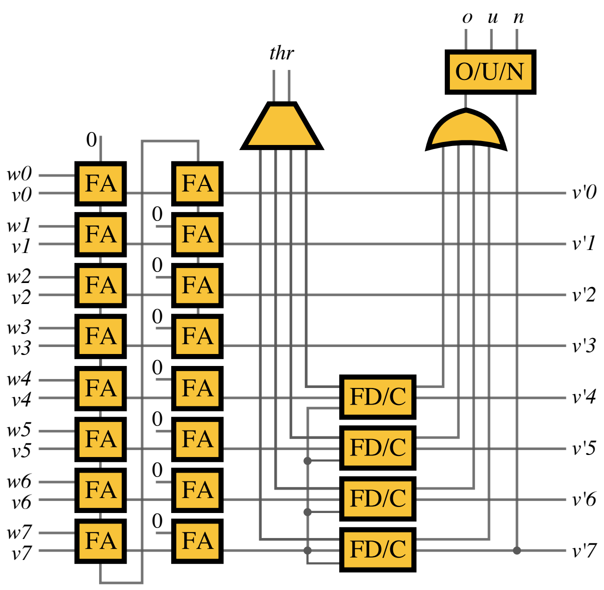


Figure A.1: Process decomposition for a 8-bit version of the Accumulator WB datapath. v, v', and w are implemented in one's complement. In synchronous design, a one's complement adder is implemented by feeding the MSB FA's carry-out to the LSB FA's carry-in. This is not possible in QDI because $(1,0)$ and $(0,1)$ inputs will not immediately produce a carry-out output, and must wait for a carry-in (which is the next bit's carry-in)–some inputs will therefore cause a deadlock. The solution is to unroll the feedback into two stages, where the first one is fed with a zero carry-in, and the second receives its carry-in from the first stage's carry-out. *thr* designates which bit correspond to the power-of-two programmable threshold. FD/C (*Flip Detect/Correct)* cells check whether the threshold bit of the sum is different from the sign bit, signifying that the value is greater than or equal to the threshold value. To subtract the threshold (correct), the bit is simply flipped back, producing v'. o, u, n (over, under, neither) is calculated by looking at v''s sign bit, and whether any FD/C detected an over-threshold event. In the implemented circuit, v is 15 bits, *thr* is 3 bits (selecting from the upper 8 bits), and w is stored as 8 bits, but is extended to 15 before being input to the adder.

Bibliography

T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith. Nengo: A python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48), 2014. ISSN 1662-5196. doi: 10.3389/fninf.2013.00048.

K. A. Boahen and A. G. Andreou. A contrast sensitive silicon retina with reciprocal synapses. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 764–772. Morgan-Kaufmann, 1992. URL `http://papers.nips.cc/paper/466-a-contrast-sensitive-silicon-retina-with-reciprocal-synapses.pdf`.

M. Boerlin, C. Machens, and S. Deneve. Predictive coding of dynamical variables in balanced spiking networks. 9:e1003258, 11 2013.

S. Choudhary, S. Sloan, S. Fok, A. Neckar, E. Trautmann, P. Gao, T. Stewart, C. Eliasmith, and K. Boahen. Silicon neurons that compute. In A. E. P. Villa, W. Duch, P. Érdi, F. Masulli, and G. Palm, editors, *Artificial Neural Networks and Machine Learning – ICANN 2012*, pages 121–128, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33269-2.

M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang. Loihi:

A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January 2018. ISSN 0272-1732. doi: 10.1109/MM.2018.112130359.

C. Eliasmith and C. H. Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, Cambridge, MA, 2003.

C. Eliasmith, T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen. A large-scale model of the functioning brain. *Science*, 338:1202–1205, 2012. doi: 10.1126/science.1225266.

D. I. Feinstein. The hexagonal resistive network and the circular approximation. Technical Report CS-TR-88-07, California Institute of Technology, 1988. URL `http://resolver.caltech.edu/CaltechCSTR:1988.cs-tr-88-07`.

S. Fok and K. Boahen. A serial h-tree router for two-dimensional arrays. In *24th IEEE International Symposium on Asynchronous Circuits and Systems*, 2018.

S. Fok, A. Neckar, and K. Boahen. Weighting and summing spike trains by accumulative thinning. In *submitted: The Thirteenth International Conference on Neuromorphic Systems (ICONS)*, submitted for publication.

D. H. Goldberg, G. Cauwenberghs, and A. G. Andreou. Probabilistic synaptic weighting in a reconfigurable network of vlsi integrate-and-fire neurons. 14:781–793, 2001.

E. Kauderer-Abrams, A. Gilbert, A. Voelker, B. Benjamin, T. C. Stewart, and K. Boahen. A population-level approach to temperature robustness in neuromorphic systems. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. doi: 10.1109/ISCAS.2017.8050985.

A. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, Pasadena, CA, 1998.

R. Manohar. Asynchronous vlsi systems. course notes, 2009.

A. J. Martin. Synthesis of asynchronous vlsi circuits. Technical Report CS-TR-93-28, California Institute of Technology, 1993. URL `http://resolver.caltech.edu/CaltechCSTR:1988.cs-tr-93-28`.

P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. ISSN 0036-8075. doi: 10.1126/science.1254642. URL `http://science.sciencemag.org/content/345/6197/668`.

A. Neckar, T. Stewart, B. Benjamin, and K. Boahen. Optimizing an analog neuron circuit design for nonlinear function approximation. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

A. R. Voelker, B. V. Benjamin, T. C. Stewart, K. Boahen, and C. Eliasmith. Extending the neural engineering framework for nonideal silicon synapses. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. doi: 10.1109/ISCAS.2017.8050810.