Analog Integrated Circuits and Signal Processing, ??, 1?? (199?)

Communicating Neuronal Ensembles between Neuromorphic Chips*

KWABENA A. BOAHEN

Physics of Computation Laboratory, MS 136-93, California Institute of Technology, Pasadena CA 91125

Received May 1, 1996. Revised ???? ?, 1996.

Editor: T. Lande

Abstract. I describe an interchip communication system that reads out pulse trains from a 64×64 array of neurons on one chip, and transmits them to corresponding locations in a 64×64 array of neurons on a second chip. It uses a random-access, time-multiplexed, asynchronous digital bus to transmit $\log_2 N$-bit addresses that uniquely identify each of the N neurons in the sending population. A peak transmission rate of 2.5MSpikes/s is achieved by pipelining the operation of the channel. I discuss how the communication channel design is optimized for sporadic stimulus-triggered activity which causes some small subpopulation to fire in synchrony, by adopting an arbitered, event-driven architecture. I derive the bandwidth required to transmit this neuronal ensemble, without temporal dispersion, in terms of the number of neurons, the probability that a neuron is part of the ensemble, and the degree of synchrony.

Keywords: random-access channel, self-timed communication, event-driven communication, arbitration, neuromorphic, retinomorphic, address-events

1. Time-division Multiplexing

The small number of input-output connections available with standard chip-packaging technology, and the small number of routing layers available in VLSI technology, place severe limitations on the degree of intra- and interchip connectivity that can be realized in multichip neuromorphic systems. Inspired by the success of time-division multiplexing in communications [1] and computer networks [2], many researchers have adopted multiplexing to solve the connectivity problem [3], [4], [5]. Multiplexing is an effective way of leveraging the 5 order-of-magnitude difference in bandwidth between a neuron (hundreds of Hz) and a digital bus (tens of megahertz), enabling us to replace dedicated point-to-point connections among thousands of neurons with a handful of high-speed connections and thousands of switches (transistors). This approach pays off in VLSI technology because transistors take up a lot less area than wires, and are becoming relatively more and more compact as the fabrication process scales down to deep submicron feature sizes.

Four important performance criteria for a communication channel that provides virtual point-to-point connections among arrays of cells are:

- **Capacity:** The maximum rate at which samples can be transmitted. It is equal to the reciprocal of the minimum communication cycle period.
- **Latency:** The mean time between sample generation in the sending population and sample reception in the receiving population.
- **Temporal Dispersion:** The standard deviation of the channel latency.
- **Integrity:** The fraction of samples that are delivered to the correct destination.

All four criteria together determine the **throughput**, which is defined as the usable fraction of the channel capacity, because the load offered to the channel must be reduced to achieve more stringent specifications for latency, temporal dispersion, and integrity.

As far as neuromorphic systems [6] are concerned, a sample is generated each time a neuronal spike occurs. These spikes carry information only in their time of occurence, since the height

* The author is presently at the University of Pennsylvania, Dept of Bioengineering, 3320 Smith Walk, Philadelphia PA 19104-6392.

and width of the spike is fixed. We must make the time it takes to communicate the occurence of each spike as short as possible, in order to maximize the throughput.

The latency of the sending neuron should not be confused with the channel latency. Neuronal latency is defined as the time interval between stimulus onset and spiking; it is proportional to the strength of the stimulus. Channel latency is an undesirable systematic offset. Similarly, neuronal dispersion should not be confused with channel dispersion. Neuronal dispersion is due to variability between individual neurons; it is inversely proportional to the strength of the stimulus. Channel dispersion is additional variability introduced by uneven access to the shared communication channel. Hence, channel latency and channel dispersion add systematic and stochastic offsets to spike times and reduce timing precision.

A growing body of evidence suggest that biological neurons have submillisecond timing precision and synchronize their firing, making it imperative to minimize channel latency and dispersion. Although neuronal transmission has been shown to be unreliable, with failure occuring at axonal branches and synapses, it is most likely that each spike changes the local state of the axon or synapse—even if it is rejected—and thereby determines the fate of subsequent spikes. So the fact that communication in the nervous system is unreliable does not give us license to build an imperfect communication channel, as the decision whether or not to transmit a spike is not arbitrary.

There are several alternatives to using the timing of fixed-width/fixed-height pulses to encode information, and several approaches to optimizing channel performance as shown in Table 1; the choices I have made are highlighted. I attempt to justify my choices by introducing a simple population activity model in Section 2. I use this model to quantify the tradeoffs faced in communication channel design in Section 3. The model assumes that the activity in the pixel array is whitened (i.e. activity is clustered in space *and* in time). Having motivated my approach to pixel-parallel communication, I describe the implementation of a pipelined communication channel, and show how a retinomorphic chip is interfaced with another neuromorphic chip in Section 4. The paper concludes with a discussion in Section 5. Parts of this work have been described previously in conference proceedings [7], [8] and in a magazine article [9].

2. Population Activity Model

Although a fairly general purpose implementation was sought, my primary motivation for developing this communication channel is to read pulse trains off a retinomorphic imager chip [7]. Therefore, the channel design was optimized for retinal population activity, and an efficient and robust solution that supports adaptive pixel-parallel quantization was sought.

2.1. Retinal Processing

The retina converts spatiotemporal patterns of incident light into spike trains. Transmitted over the optic nerve, these discrete spikes are converted back into continuous signals by dendritic integration of excitatory postsynaptic potentials in the lateral geniculate nucleus of the thalamus. For human vision, contrast thresholds of less than 1%, processing speeds of about 20 ms per stage, and temporal resolution in the submillisecond range are achieved, with spike rates as low as a few hundred per second. No more than 10 spikes per input are available during this time. The retina achieves such high peprformance by minimizing redundancy and maximizing the information carried by each spike.

The retina must encode stimuli generated by all kinds of events efficiently, over a large range of lighting conditions and stimulus velocities. These events fall into three broad classes, listed in order of decreasing probability of occurrence:

1. *Static events:* Generate stable, long-lived stimuli; examples are buildings or trees in the backdrop.
2. *Punctuated events:* Generate brief, short-lived stimuli; examples are a door opening, a light turning on, or a rapid, short saccade.
3. *Dynamic events:* Generate time-varying, ongoing stimuli; examples are a spinning wheel, grass vibrating in the wind, or a smooth-pursuit eye movement.

In the absence of any preprocessing, the output activity mirrors the input directly. Changes

Table 1. Time-Multiplexed Communication Channel Design Options. The choices made in this work are highlighted.

Specification	Approaches	Remarks
Activity Encoding	Pulse Amplitude	Long settling time and static power dissipation
	Pulse Width	Channel capacity degrades with increasing width
	Pulse Code	Inefficient at low precision ($<$ 6 bits)
	Pulse Timing	**Uses minimum-width, rail-to-rail pulses**
Latency	Polling	Latency $\propto$ Total number of neurons
	Event-driven	**Latency $\propto$ Number currently active**
Integrity	Collision Rejection	Collisions increase exponentially with throughput
	Arbitration	**Reorder events to prevent collisions**
Dispersion	Dumping	New events are given priority $\Rightarrow$ No dispersion
	Queueing	**Dispersion $\propto$ 1/capacity, at constant throughput**
Capacity	Hard-wired	Simple $\Rightarrow$ Short cycle time
	Pipelined	**Cycle time set by a single stage**

in illumination, which influence large areas, are reflected directly in the output of every single pixel in the region affected. Static events, such as a stable background, generate persistent activity in a large fraction of the output cells, which transmit the same information over and over again. Punctuated events generate little activity and are transmitted without any urgency. Dynamic events generate activity over areas far out of proportion to informative features in the stimulus, when the stimulus rapidly sweeps across a large region of the retina. Clearly, these output signals are highly correlated over time and over space, resulting in a high degree of redundancy. Hence, reporting the raw intensity values makes poor use of the limited throughput of the optic nerve.

The retina has evolved exquisite filtering and adaptation mechanisms to improve coding efficiency, six of which are described briefly below[10], [11]:

1. *Local automatic gain control* at the receptor level eliminates the dependence on lighting—the receptors respond to only contrast—extending the dynamic range of the retina's input without increasing its output range.
2. *Bandpass spatiotemporal filtering* in the first stage of the retina (outer plexiform layer or OPL) attenuates signals that do not occur at a fine spatial *or* a fine temporal scale, ameliorating redundant transmission of low frequency signals and eliminating noisy high frequency signals.
3. *Highpass temporal and spatial filtering* in the second stage of the retina (inner plexiform layer or IPL) attenuates signals that do not occur at a fine spatial scale *and* a fine temporal scale, eliminating the redundant signals passed by the OPL, which responds strongly to low temporal frequencies that occur at high spatial frequencies (sustained response to static edge) or to low spatial frequencies that occur at high temporal frequencies (blurring of rapidly moving edge).
4. *Half-wave rectification*, together with dual-channel encoding (ON and OFF output cell types), in the relay cells between the OPL and the IPL (bipolar cells) and the retina's output cells (ganglion cells) eliminates the elevated quiescent neurotransmitter release rates and firing rates required to signal both positive and negative excursions using a single channel.
5. *Phasic transient–sustained response* in the ganglion cells avoids temporal aliasing by transmitting rapid changes in the signal using a brief, high frequency burst of spikes, and, at the same time, avoids redundant sampling, by transmitting slow changes in the signal using modulation of a low sustained firing rate.
6. *Foveated architecture*, together with actively directing the gaze (saccades), eliminates the need to sample all points in the scene at the highest spatial and temporal resolution, while providing the illusion of doing so everywhere. The cell properties are optimized: smaller and more sustained in the fovea (parvocellular or X cell type), where the image is stabilized by tracking,

and larger and more transient in the periphery (magnocellular or Y cell type), where motion occurs.

The resulting activity in the ganglion cells, which convert these preprocessed signals to spikes, and transmit the spikes over the optic nerve, is rather different from the stimulus pattern. For relatively long periods, the scene captured by the retina is stable. These static events produce sparse activity in the OPL's output, since the OPL does not respond to low spatial frequencies, and virtually no activity in the IPL's output, since the IPL is selective for temporal frequency as well as spatial frequency. The OPL's sustained responses drive the 50,000, or so, ganglion cells in the fovea, allowing the fine details of the stabilized object to be analyzed. While the vast majority of the ganglion cells, about 1 million in all, are driven by the IPL, and fire at extremely low quiescent rates of 10S/s (spikes per sec), or less, in response to the static event.

When a punctuated event (e.g. a small light flash) occurs, the IPL responds strongly, since both high temporal frequencies and high spatial frequencies are present, and a minute subpopulation of the ganglion cells raise their firing rates, briefly, to a few hundred spikes per second. A dynamic event, such as a spinning windmill, or a flash that lights up an extended region, is effectively equivalent to a sequence of punctuated events occurring at different locations in rapid succession, and can potentially activate ganglion cells

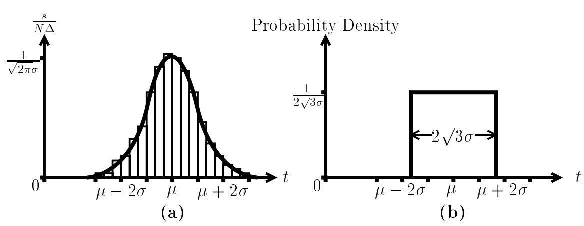


Fig. 1. The hypothetical distribution, in time, of samples generated by a subpopulation of neurons triggered by the same stimulus at $t = 0$. (a) Normalized poststimulus time histogram with Gaussian fit: s is the number of samples per bin, Δ is the bin-width, and N is the total number of samples; μ and σ are the mean and standard deviation of the fitted Gaussian. (b) Rectangular distribution with the same mean and standard deviation as the Gaussian. After redistributing samples uniformly, the sampling rate is only 72% of the peak rate reached in the original distribution, yielding the minimum capacity required to transmit the burst without dispersion.

over an extended region simultaneously. This is indeed the case for the OPL-driven cells but is not true for the IPL-driven cells, which cover most of the retina, because the low spatial frequencies produced in the OPL's output by such a stimulus prevent the IPL from responding.

In summary, the activity in the optic nerve is clustered in space *and* time (whitened spectrum), consisting of sporadic short bursts of rapid firing, triggered by punctuated and dynamic events, overlaid on a low, steady background firing rate, driven by static events.

2.2. The Neuronal Ensemble

We can describe the activity of a neuronal population by an ordered list of locations in spacetime

$$\mathcal{E} = \{(x_0; t_0), (x_1; t_1), \ldots (x_i; t_i), \ldots\}; \quad t_0 < t_1 < \cdots t_i < \cdots,$$

where each coordinate specifies the occurrence of a spike at a particular location, at a particular time. The same location can occur in the list several times but a particular time can occur only once—assuming time is measured with infinite resolution.

There is no need to record time explicitly if the system that is logging this activity operates on it in real-time—only the location is recorded and time represents itself. In that case, the representation is simply

$$\mathcal{E} = \{x_0, x_1, \ldots x_i, \ldots\}; \ t_0 < t_1 < \cdots t_i < \cdots.$$

This real-time representation is called the *address-event representation* (AER) [5], [4]. I shall present more details about AER later. At present, my goal is to develop a simple model of the probability distribution of the neuronal population activity described by $\mathcal{E}$.

$\mathcal{E}$ has a great deal of underlying structure that arises from events occurring in the real world, to which the neurons are responding. The elements of $\mathcal{E}$ are clustered at temporal locations where these events occur, and are clustered at spatial locations determined by the shape of the stimulus. Information about stimulus timing and shape can therefore be obtained by extracting these clusters. $\mathcal{E}$ also has an unstructured component that arises from noise in the signal, from noise in the system, and from differences in gain and state among the neurons. This stochastic component limits the precision with which the neurons can encode information about the stimulus.

We can use a much more compact probabilistic description for $\mathcal{E}$ if we know the probability distributions of the spatial and temporal components of the noise. In that case, each cluster can be described explicitly by its mean spatial configuration and its mean temporal location, and the associated standard deviations.

$$\begin{aligned}\mathcal{E} \;\simeq\; &\{(\overline{x}_0, \sigma_{x0}; \overline{t}_0, \sigma_{t0}), (\overline{x}_1, \sigma_{x1}; \overline{t}_1, \sigma_{t1}),\\ &\ldots, (\overline{x}_i, \sigma_{xi}; \overline{t}_i, \sigma_{ti}), \ldots, \};\\ &t_0 < t_1 \cdots t_i < \cdots.\end{aligned}$$

I shall call these statistically-defined clusters *neuronal ensembles*. The ith neuronal ensemble is defined by a probability density function $p_i(x; t)$

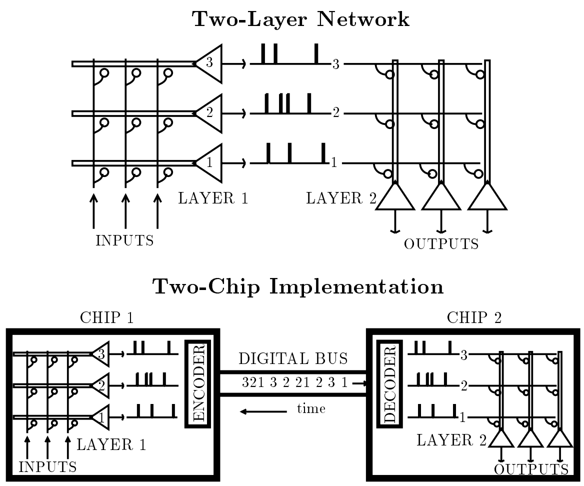


Fig. 2. Two-chip implementation of a two-layer spiking neural network (Adapted from [4]). In the two-layer network, the origin of each spike is infered from the line on which it arrives. This parallel transmission uses a labeled-line representation. In the two-chip implementation, the neurons share a common set of lines (digital bus) and the origin of each spike is preserved by broadcasting one of the labels. This serial transmission uses an *address-event representation* (AER). No information about the height or width of the spike is transmitted, thus information is carried by spike timing only. A shared address-event bus can transparently replace dedicated labeled lines if the encoding, transmission, and decoding processes cycle in less than Δ/n seconds, where Δ is the desired spike timing precision and n is the maximum number of neurons that are active during this time.

with parameters $\overline{x}_i$, σ_{xi}, $\overline{t}_i$, σ_{ti}. The probability that a spike generated at location x at time t is a member of the ith ensemble is then obtained by computing $p_i(x; t)$.

In what follows, I will assume that the distribution of the temporal component of the noise is Gaussian and the latency (the mean minus the stimulus onset time) and the standard deviation are inversely proportional to the stimulus strength. We can measure the parameters of this distribution by fitting a Gaussian to the normalized poststimulus time histogram, as shown in Figure 1a. Here, latency refers to the time it takes to bring a neuron to spiking threshold; this neuronal latency is distinct from the channel latency that I defined earlier. The Gaussian distribution arises from independent random variations in the characteristics of these neurons—whether carbon or silicon based—as dictated by the Central Limit Theorem.

The ratio between the standard deviation and the latency, which is called the *coefficient of variation* (cov), is used here as a measure of the variability among the neurons. Other researchers have used the cov to measure mismatch among transistors and the variability in the steady firing rates of neurons; I use it to measure variability in latency across a neuronal ensemble. As the neuronal latency decreases, the neuronal temporal dispersion decreases proportionally, and hence the cov remains constant. The cov is constant because the characteristics that vary between individual neurons, such as membrane capacitance, channel conductance, and threshold voltage, determine the slope of the latency versus input current curves—rather than the absolute latency or firing rate.

3. Design Tradeoffs

Several options are available to the communication channel designer. Should he preallocate the channel capactiy, giving a fixed amount to each node, or allocate capacity dynamically, matching each nodes allocation to its current needs? Should she allow the users to transmit at will, or implement elaborate mechanisms to regulate access to the channel? And how does the distribution of activity over time and over space impact these choices? Can he assume that nodes act randomly, or are there significant correlations between the activities of these nodes? I shed some light on these questions in this section, and provide some definitive answers.

3.1. Allocation: Dynamic or Static?

We are given a desired sampling rate f_{Nyq}, and an array of N signals to be quantized. We may use adaptive, 1-bit, quantizers that sample at f_{Nyq} when the signal is changing, and sample at f_{Nyq}/Z when the signal is static. Let the probability that a given quantizer samples at f_{Nyq} be a. That is, a is the fraction of the quantizers whose inputs are changing. Then, each quantizer generates bits at the rate

$$f_{\mathrm{bits}} = f_{\mathrm{Nyq}}(a + (1-a)/Z)\log_2 N,$$

because a percent of the time, it samples at f_{Nyq}; the remaining $(1-a)$ percent of the time, it samples at f_{Nyq}/Z. Furthermore, each time that it samples, $\log_2 N$ bits are sent to encode the location, using the aforemention address-event representation (AER) [5], [4]. AER is a fairly general scheme for transmitting information between arrays of neurons on separate chips, as shown in Figure 2.

On the other hand, we may use more conventional qunatizers that sample every location at f_{Nyq}, and do not locally adapt the sampling rate. In that case, there is no need to encode location explicitly; we simply cycle through all N locations, according to a fixed squence order, and infer the origin of each sample from its position in the sequence. For a constant sampling rate, the bit rate per quantizer is simply f_{Nyq}.

Adaptive sampling produces a lower bit rate than fixed sampling if

$$a < (Z/(Z-1))(1/\log_2 N - 1/Z).$$

For example, in a 64×64 array with sampling-rate attenuation, Z, of 40, the active fraction, a, must be less than 6.1 percent. In a retinomorphic system, the adaptive neuron circuit performs sampling-rate attenuation (Z will be equal to the firing rate attenuation factor γ) [10], and the spatiotemporal bandpass filter makes the output activity sparse [10], resulting in a low active fraction a.

It may be more important to minimize the number of samples produced per second, instead of minimizing the bit rate, as there are usually sufficient I/O pins to transmit all the bits in each sample in parallel. In that case, it is the number of samples per second that is fixed by the channel capacity.

Given a certain fixed channel throughput (F_{chan}), in samples per second, we may compare the effective sampling rates, f_{Nyq}, acheived by the two strategies. For adaptive quantization, channel throughput is allocated dynamically in the ratio $a : (1-a)/Z$ between the active and passive fractions of the node population. Hence,

$$f_{\mathrm{Nyq}} = f_{\mathrm{chan}}/(a + (1-a)/Z) \tag{1}$$

where $f_{\mathrm{chan}} \equiv F_{\mathrm{chan}}/N$ is the throughput per node. In contrast, fixed quantization achieves only f_{chan}. For instance, if $f_{\mathrm{chan}} = 100$S/s, adaptive quantization achieves $f_{\mathrm{Nyq}} = 1.36$KS/s, with an active fraction of 5 percent and a sampling-rate attenuation factor of 40. Thus, a fourteen-fold increase in temporal bandwidth is achieved under these conditions; the channel latency also is reduced by the same factor.

3.2. Access: Arbitration or Free-for-All?

If we provide random access to the shared communication channel, in order to support adaptive pixel-level quantization [9], we have to deal with contention for channel access, which occurs when two or more pixels attempt to transmit simultaneously. We can introduce an arbitration mechanism to resolve contention and a queueing mechanism to allow nodes to wait for their turn. However, arbitration lengthens the communication cycle period, reducing the channel capacity, and queuing causes temporal dispersion, corrupting timing information. On the other hand, if we simply allow collisions to occur, and discard the corrupted samples so generated [12], we may achieve a shorter cycle period and reduced dispersion, but sample loss will increase as the load increases.

We may quantify this tradeoff using the following well-known result for the collision probability [1]:

$$p_{\mathrm{coll}} = 1 - e^{-2G}, \tag{2}$$

where G is the offered load, expressed as a fraction of the channel capacity.[1] For $G < 1$, G is the probability that a sample is generated during the communication cycle period.

If we arbitrate, we will achieve a certain cycle time, and a corresponding channel capacity, in a given VLSI technology. An arbitered channel can operate at close to 100-percent capacity because the 0.86 collision probability for $G = 1$ is not a problem—users just wait their turn. Now, if we do not arbitrate, we will achieve a shorter cycle time, with a proportionate increase in capacity. Let us assume that the cycle time is reduced by a factor of 10, which is optimistic. For the same offered load, we have $G = 0.1$, and find that $p_{\mathrm{coll}} = 18$ percent. Thus, the simple nonarbitered channel can handle more spikes per second only if collision rates higher than 18 percent are acceptable. For lower collision rates, the complex, arbitered channel offers more throughput, even though its cycle period is 1 order of magnitude longer, because the nonarbitered channel can utilize only 10 percent of its capacity.

Indeed, the arbiterless channel must operate at high error rates to maximize utilization of the channel capacity. The throughput is Ge^{-2G} [1], since the probability of a successful transmission (i.e., no collision), is e^{-2G}. Throughput reaches a maximum when the success rate is 36 percent (e^{-1}) and the collision rate is 64 percent. At maximum throughput, the load, G, is 50 percent and, hence, the peak channel throughput is only 18 percent. Increasing the load beyond the 50% level lowers the channel utilization because the success rate falls more rapidly than the load increases.

In summary, the simple, free-for-all design offers higher throughput if high data-loss rates are tolerable, whereas the complex, arbitered design offers higher throughput when low-data loss rates are desired. And, due to the fact that the maximum channel untilization for the free-for-all channel is only 18 percent, the free-for-all channel will not be competitive at all unless it can achieve a cycle time that is five times shorter than that of the arbitered channel.

Indeed, the free-for-all protocol was first developed at the University of Hawaii in the 1970s to provide multiple access between computer terminals and a time-shared mainframe over a wireless link with an extremely short cycle time; it is known as the ALOHA protocol [1]. In this application, a vanishingly small fraction of the wireless link's tens of megahertzs of bandwidth is utilized by people typing away at tens of characters per second on a few hundred computer terminals, and hence the error rates are negligible. However, in a neuromorphic system where we wish to service hundreds of thousands of neurons, efficient utilization of the channel capacity is of paramount concern.

The inefficiency of ALOHA has been long recognized, and researchers have developed more efficient communication protocols. One popular approach is CSMA (carrier sense, multiple access), where each user monitors the channel and does not transmit when the channel is busy. The Ethernet protocol uses this technique, and most local-area-networks work this way. Making available information about the state of the channel to all users greatly reduces the number of collisions. A collision occurs only when two users attempt to transmit within a time interval that is shorter than the time it takes to update the information about the channel state. Hence, the collision rate drops if the time that users spend transmitting data is longer than the round trip delay; if not, CSMA's performance is no better than ALOHA's [1]. Bassen's group is developing a CSMA-like protocol for neuromorphic communication to avoid the queueing associated with arbitration [13].

What about the timing errors introduced by queueing in the arbitered channel? It is only fair to ask whether these timing errors are not worse than the data-loss errors. The best way to make this comparison is to express the channel's latency and temporal dispersion as fractions of the neuronal latency and temporal dispersion, respectively. If the neuron fires at a rate $f_{\rm Nyq}$, we may assume, for simplicity, that its latencies are uniformly distributed between 0 and $T_{\rm Nyq} = 1/f_{\rm Nyq}$. Hence, the neuronal latency is $\mu = T_{\rm Nyq}/2$, and the neuronal temporal dispersion $\sigma = T_{\rm Nyq}/(2\sqrt{3})$. For this flat distribution, the coefficient of variation is $c = 1/\sqrt{3}$, which equals 58 percent.

To find the latency and temporal dispersion introduced by the queue, we use a well-known result from queueing theory which gives the moments of the time spent waiting in the queue, $\overline{w^n}$, as a function of the moments of the service time, $\overline{x^n}$ [14]:

$$\begin{aligned} \overline{w} &= \frac{\lambda\overline{x^2}}{2(1-G)}, \\ \overline{w^2} &= 2\overline{w}^2 + \frac{\lambda\overline{x^3}}{3(1-G)}; \end{aligned}$$

where λ is the arrival rate of the samples.[2] An interesting property of the queue, which is evident from these results, is that the first moment of the waiting time increases linearly with the second moment of the service time. Similarly, the second moment of the waiting time increases linearly with the third moment of the service time.

In our case, we may assume that the service time, Δ, is fixed; hence $\overline{x^n} = \Delta^n$. In that case, the mean and the variance of the number of cycles spent waiting are given by

$$\overline{m} \equiv \frac{\overline{w}}{\Delta} = \frac{G}{2(1-G)}, \tag{3}$$

$$\sigma_m^2 = \frac{\overline{w^2} - \overline{w}^2}{\Delta^2} = \overline{m}^2 + \frac{2}{3}\overline{m}. \tag{4}$$

For example, at 95-percent capacity, a sample spends 9.5 cycles in the queue, on average. This result agrees with intuition: As every twentieth slot is empty, one must wait anywhere from 0 to 19 cycles to be serviced, which averages out to 9.5. Hence the latency is 10.5 cycles, including the additional cycle required for service, and the temporal dispersion is 9.8 cycles—virtually equal to the latency. In general, the temporal dispersion will be approximately equal to the latency whenever the latency is much larger than one cycle.

If there are a total of N neurons, the cycle time is $\Delta = G/(N f_{\rm chan})$, where G is the normalized load, and the timing error due to channel latency will be

$$e_\mu \equiv \frac{(m+1)\Delta}{\mu} = 2G\frac{f_{\rm Nyq}}{f_{\rm chan}}\frac{m+1}{N}$$

Using the expression for the number of cycles spent waiting (Equation 3), and the relationship between $f_{\rm Nyq}$ and $f_{\rm chan}$ (Equation 1), we obtain

$$e_\mu = \frac{2G}{N}\frac{1}{a+(1-a)/Z}\frac{1-G/2}{1-G}.$$

For example, at 95 percent load and at 5 percent active fraction, with a sampling rate attenuation of 40 and with a population of size 4096 (64×64), the latency error is 7 percent. The error introduced by the temporal dispersion in the channel will be similar, as the temporal dispersion is more or less equal to the latency.

Notice that the timing error is inversely proportional to N. This scaling occurs because channel capacity must grow with the number of neurons. Hence, the cycle time decreases, and there is a proportionate decrease in queueing time, even though the number of cycles spent queueing remains the same—for the same normalized load. In contrast, the collision rate remains unchanged for the same normalized load. Hence, the arbitered channel scales much better than the nonarbitered one as technology improves and shorter cycle times are achieved.

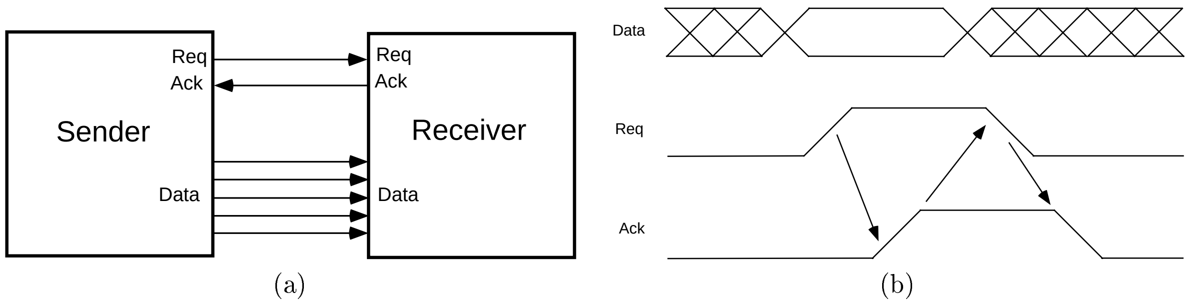


Fig. 3. Self-timed data-transmission protocol using a four-phase handshake. (a) Data-bus (data) and data-transfer control signals (Req and Ack). (b) Handshake protocol on control lines. The sender initiates the sequence by driving its data onto the bus and taking Req high. The receiver reads the data when Req goes high, and drives Ack high when it is done. The sender widthdraws its data and takes Req low when Ack goes high. The receiver terminates the sequence by taking Ack low after Req goes low, returning the bus to its original state. As data is latched on Ack $\uparrow$, the designer can ensure that the setup and hold times of the receiver's input latch are satisfied by delaying Req $\uparrow$, relative to the data, and by delaying widthdrawing the data after Ack $\uparrow$.

3.3. *Traffic: Random or Correlated?*

Correlated spike activity occurs when external stimuli trigger synchronous activity in several neurons. Such structure, which is captured by the neuronal ensemble concept, is much more plausible than totally randomized spike times—especially if neurons are driven by sharply defined object features (high spatial and temporal frequencies) and adapt to the background (low spatial and temporal frequencies). If there are correlations among firing times, the Poisson distribution does not apply to the spike times, but, making a few reasonable assumptions, it may be used to describe the relative timing of spikes within a burst.

The distribution of sample times *within* each neuronal ensemble is best described by a Gaussian distribution, centered at the mean time of arrival, as shown in Figure 1a. The mean of the Gaussian, μ, is taken to be the delay between the time the stimulus occurred and the mean sample time, and the standard deviation of the Gaussian, σ, is assumed to scale with the mean, i.e. the coefficient of variation (cov) $c \equiv \sigma/\mu$, is constant.

The minimum capacity, per quantizer, required to transmit a neuronal ensemble without temporal dsipersion is given by

$$f_{\text{burst}} \equiv \frac{F_{\text{burst}}}{N_{\text{burst}}} = \frac{1}{2\sqrt{3}c\mu},$$

using the equivalent uniform distribution shown in Figure 1b. This simple model predicts that shorter latencies or less variability can be had only by paying for a proportionate increase in channel throughput. For instance, a latency of 2ms and a cov of 10% requires 1440 S/s per quantizer. This result assumes that the neurons' interspike intervals are large enough that there is no overlap in time between successive bursts; this is indeed the case if $c < 1/\sqrt{3}$ or 58%.

There is a strong tendency to conclude that the minimum throughput specification is simply equal to the mean sampling rate. This is the case only if sampling times are totally random. Random samples are distributed uniformly over the period $T = 1/f$, where f is the quantizer's mean sampling rate, assumed to be the same for all nodes that are part of the ensemble; the latency is $\mu = T/2$, and the temporal dispersion is $\sigma = T/(2\sqrt{3})$. Hence, the cov is $c = 1/\sqrt{3}$ =58% and Equation 3.3 yields $f_{\text{burst}} = 1/T$, as expected. For a latency of 2ms and a cov of 58%, the required throughput is 250 S/s per quantizer compared to 1440 S/s when the cov is 10%.

The rectangular (uniform) approximation to the Gaussian, shown in Figure 1b, may be used to calculate the number of collisions that occur during a burst. We simply use Equation 2, and set the rate of the Poisson process to the uniform sampling rate of the rectangular distribution. The result is plotted in Figure 9.

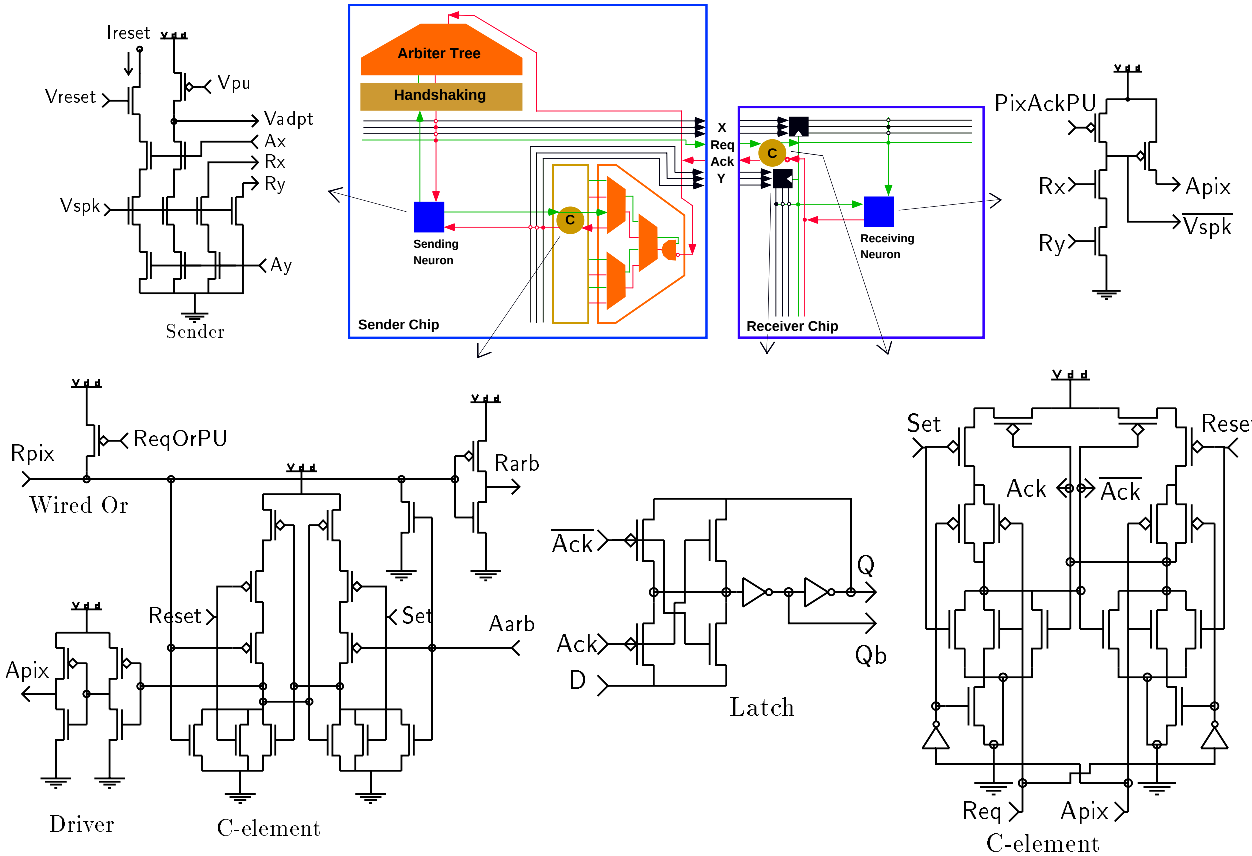


Fig. 4. Pipelined addess-event channel. The block diagram describes the channel architecture; the logic circuits for each block also are shown. Sender Chip: The row and column arbiter circuits are identical; the row and column handshaking circuits (C-elements) are also identical. The arbiter is built from a tree of two-input arbiter cells that send a request signal to, and receive a select signal from, the next level of the tree. The sending neuron's logic circuit (upper-left) interfaces between the adaptive neuron circuit, the row C-element (lower-left: Ry → Rpix, Apix → Ay), and the column C-element (lower-left: Rx → Rpix, Apix → Ax). The pull-down chains in the pixel—tied to pull-up elements at the right and at the top of the array—and the column and row request lines form wired-OR gates. The C-elements talk to the column and row arbiters (detailed circuit not shown) and drive the address encoders (detailed circuit not shown). The encoders generate the row address (Y), the column address (X), and the chip request (Req). Receiver Chip: The receiver's C-element (lower-right) acknowledges the sender, strobes the latches (lower-middle), and enables the address decoders (detailed circuit not shown). The receiver's C-element also monitors the sender's request (Req) and the receiving neuron's acknowledge (Apix). The receiving neuron's logic circuit (upper-right) interfaces between the row and column selects (Ry and Rx) and the post-synaptic integrator, and generates the receiving neuron's acknowledge (Apix). The pull-down path in the pixel—tied to pull-up elements at the left of the array—and the row acknowledge lines form wired-OR gates. An extra wired-OR gate, that runs up the left edge of the array, combines the row acknowledges into a single acknowledge signal that goes to the C-element.

3.4. Throughput Requirements

By including the background activity of the neurons that do not participate in the neuronal ensemble, we obtain the total throughput requirement

$$f_{\rm total} = a f_{\rm burst} + (1-a) f_{\rm fire}/Z,$$

where a is the fraction of the population that participates in the burst; $f_{\rm fire}/Z$ is the firing rate of the remaining quantizers, expressed as an attenuation, by the factor Z, of the sampling rate of the active quantizers. Assuming $f_{\rm fire} = 1/(2\mu)$, we have $f_{\rm fire}/f_{\rm burst} = \sqrt{3c}$, and

$$f_{\rm total} = \frac{a + \sqrt{3c}(1-a)/Z}{2\sqrt{3c}\mu},$$

per quantizer. For a 2ms latency, a 10 percent cov, a 5 percent active fraction, and an attenuation factor of 40, the result is $f_{\rm total} = 78.1$S/s per quantizer. For these parameter values, a 15.6 percent temporal dispersion error is incurred, assuming a channel loading of 95 percent and a population size of 4096 (64 by 64 array).

This result is only valid if samples are not delayed for more than the duration of the burst (i.e. errors less than $1/\sqrt{3} = 58\%$). For larger delays, the temporal dispersion grows linearly—instead of hyperbolically—because the burst is distributed over an interval no greater than $(\hat{F}_{\rm burst}/F_{\rm chan})/T_{\rm burst}$, when the sample rate $\hat{F}_{\rm burst}$ exceeds the channel capacity, $F_{\rm chan}$, where $T_{\rm burst}$ is the duration of the burst.

4. Pipelined Communication Channel

In this section, I describe an arbitered, random-access communication channel design that supports asynchronous pixel-level analog-to-digital conversion. As discussed in the previous section, arbitration is the best choice for neuromorphic systems whose activity is sparse in space and in time, because it allows us to trade an exponential increase in collisions for a linear increase in temporal dispersion. Furthermore, for the same percentage channel utilization, the temporal dispersion decreases as the technology improves, and we build larger systems with shorter cycle times, whereas the collision probability remains the same.

The downside of arbitration is that this process lengthens the communication cycle, reducing channel capacity. I have achieved improvements in throughput over previous arbitered designs [4] by adopting three strategies that shorten the average cycle time:

1. *Allow several address-events to be in various stages of transmission at the same time.* This well-known approach to increasing throughput is called pipelining; it involves breaking the communication cycle into a series of steps and overlapping the execution of these steps as much as possible.
2. *Exploit locality in the arbiter tree.* That is, do not arbitrate among all the inputs every time; doing so would require spanning all $\log_2(N)$ levels of the tree. Instead, find the smallest subtree that has a pair of active inputs, and arbitrate between those inputs; this approach minimizes the number of levels spanned.
3. *Exploit locality in the row–column architecture.* That is, do not redo both the row arbitration and the column arbitration for each address-event. Instead, service all requesting pixels in the selected row, redoing only the column arbitration, and redo the row arbitration only when no more requests remain in the selected row.

This work builds on the pioneering contributions of Mahowald [4] and Sivilotti [5]. Like their original design, my implementation is completely self-timed: Every communication consists of a full four-phase handshaking sequence on a pair of wires, as shown in Figure 3. Self-timed operation makes queueing and pipelining straightforward [15]: You stall a stage of the pipeline or make a pixel wait simply by refusing to acknowledge it. Lazzaro et.al. have also improved on the original design [16], and have used their improved interface in a silicon auditory model [17].

4.1. Communication Cycle Sequence

The operations involved in a complete communication cycle are outlined in this subsection. This description refers to the block diagram of the channel architecture in Figure 4; the circuits are described in the next two subsections. At the beginning of a communication cycle, the request and acknowledge signals are both low.

On the sender side, spiking neurons first make requests to the Y arbiter, which selects only one row at a time. All spiking neurons in the selected row then make requests to the X arbiter. At the same time, the Y address encoder drives the address of the selected row onto the bus. When the X arbiter selects a column, the neuron in that particular column, and in the row selected earier, resets itself and withdraws its column and row requests. At the same time, the X address encoder drives the addresses of the selected column on to the bus, and takes Req high.

When Ack goes high, the select signals that propagate down the arbiter tree are disabled by the AND gates at the top of the X and Y arbiters. As a result, the arbiter inactivates the select signals sent to the pixels and to the address-encoders. Consequently, the sender withdraws the addresses and the request signal Req.

When it is necessary, the handshake circuit (also known as a C-element [15]) between the arbiters and the rows or columns will delay inactivating the select signals that drive the pixel, and the encoders, to give the sending pixel enough time to reset. The sender's handshake circuit is designed to stall the communication cycle by keeping Req high until the pixel withdraws its row and column requests, confirming that the pixel has reset. The exact sequencing of these events is shown in Figure 10.

On the receiver side, as soon as Req goes high, the address bits are latched and Ack goes high. At the same time, the address decoders are enabled and, while the sender chip is deactivating its internal request and select signals, the receiver decodes the addresses and selects the corresponding pixel. When the sender takes Req low, the receiver responds by taking Ack low, disabling the decoders and making the latches transparent again.

When it is necessary, the receiver's handshake circuit, which monitors the sender's request (Req) and the acknowledge from the receiving pixel (Apix), will delay disabling the address-decoders to give the receiving pixel enough time to read the spike and generate a post-synaptic potential. The reciever's handshake circuit is designed to stall the communication cycle by keeping Ack high until the pixel acknowledges, confirming that the pixel did indeed recieve the spike. The exact sequencing of these events also is shown in Figure 10.

4.2. Arbiter Operation

The arbiter works in a hierarchical fashion, using a tree of two-way decision cells [5], [4], [16], as shown in Figure 4. Thus, arbitration between N inputs requires only $N-1$ two-input cells. The N-input arbiter is layed out as a $(N-1)\times 1$ array of cells, positioned along the edge of the pixel array, with inputs from the pixels coming in on one side and wiring between the cells running along the other side.

The core of the two-input arbiter cell is a flip-flop with complementary inputs and outputs, as shown in Figure 5; these circuits were designed by Sivilotti and Mahowald [5], [4]. That is, both the set and reset controls of the flip-flop (tied to R1 and R2) are normally active (i.e., low), forcing both of the flip-flops outputs (tied to Q1 and Q2) to be active (i.e., high). When one of the two incoming requests (R1 or R2) becomes active, the corresponding control (either set or reset) is inactivated, and that request is selected when the corresponding output (Q1 or Q2) becomes inactive. In case both of the cell's incoming requests become active simultaneously, the flip-flop's set and reset controls are both inactivated, and the flip-flop randomly settles into one of its stable states, with one output active and the other inactive. Hence, only one request is selected.

Before sending a select signal (A1 or A2) to the lower level, however, the cell sends a request signal (Rout) up the tree and waits until a select signal (Ain) is received from the upper level. At the top of the tree, the request signal is simply fed back in, and becomes the select signal that propagates down the tree.

As the arbiter cell continues to select a branch so long as there is an active request from that branch, we can keep a row selected, until all the active pixels in that row are serviced, simply by ORing together all the requests from that row to generate the request to the Y arbiter. Similarly, as the request passed to the next level of the tree is simply the OR of the two incoming requests, a subtree will remain selected as long as there are active requests in that part of the arbiter tree. Thus, each subtree will service all its daughters once it is selected. Using the arbiter in this way minimizes the number of levels of arbitration performed—the input that requires the smallest number of levels to be crossed is selected.

To reset the select signals fed into the array—and to the encoder—previous designs removed the in-coming requests at the bottom of the arbiter tree [5], [4], [16]. Hence, the state of all the flip-flops were erased, and a full $\log_2(N)$-level row arbitration and a full column arbitration had to be performed for every cycle. In my design, I reset the row/column select signals by removing the select signal from the top of the arbiter tree; thus, the request signals fed in at the bottom are undisturbed, and the state of arbiter is preserved, al-

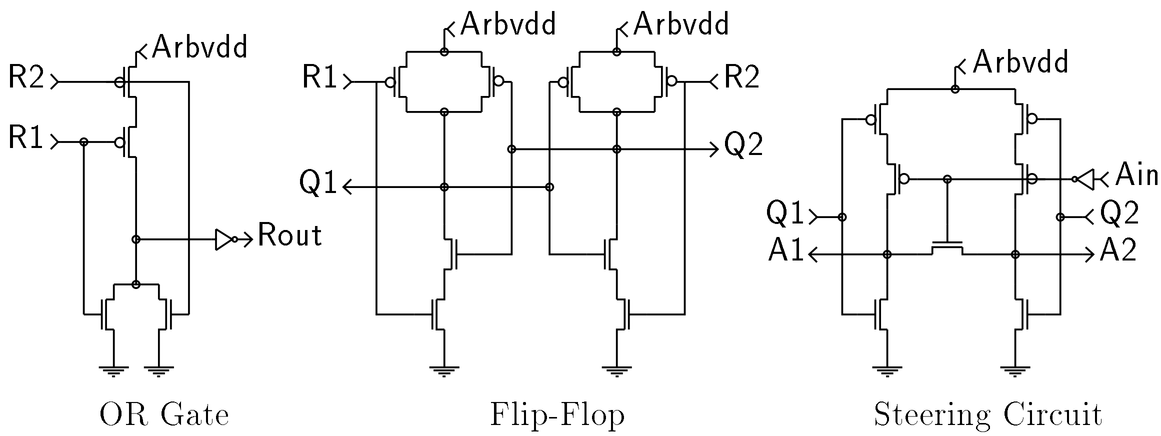


Fig. 5. Arbiter cell circuitry. The arbiter cell consists of an OR gate, a flip-flop, and a steering circuit. The OR gate propagates the two incoming active-high requests, R1 and R2, to the next level of the tree by driving Rout. The flip-flop is built from a pair of cross-coupled NAND gates. Its active-low set and reset inputs are driven by the incoming requests, R1 and R2, and its active-high outputs, Q1 and Q2, control the steering circuit. The steering circuit propagates the incoming active-high select signal, Ain, down the appropriate branch of the tree by driving either A1 or A2, depending on the state of the flip-flop. This circuitry is reproduced from Mahowald 1994 [4].

lowing locality in the array and the arbiter tree to be fully exploited.

I achieve a shorter average cycle time by exploiting locality; this opportuinistic approach trades fairness for efficiency. Instead of allowing every active pixel to bid for the next cycle, or granting service on a strictly first-come–first-served basis, I take the travelling-salesman approach, and service the customer that is closest. Making the average service time as short as possible—to maximize channel capacity—is my paramount concern, because the wait time goes to infinity when the channel capacity is exceeded.

4.3. *Logic Circuits and Latches*

In this subsection, I describe the address-bit latches and the four remaining asynchronous logic gates in the communication pathway. Namely, the interfaces in the sending and receiving neurons and the C-elements in the sender and receiver chips. The interactions between these gates, the neurons, and the arbiter—and the sequencing constraints that these gates are designed to enforce—are depicted graphically in Figure 10.

The logic circuit in the sending neuron is shown in Figure 4; it is similar to that described in [4], [16]. The neuron takes Vspk high when it spikes, and pulls the row request line Ry low. The column request line Rx is pulled low when the row select line Ay goes high, provided Vspk is also high. Finally, Ireset is turned on when the column select line Ax goes high, and the neuron is reset. Vadpt is also pulled low to dump charge on the feedback integrator in order to adapt the firing rate [18].

I added a third transistor, driven by Vspk, to the reset chain to turn off Ireset as soon as the neuron is reset, i.e. Vspk goes low. Thus, Ireset does not continue to discharge the input capacitance while we are waiting for Ax and Ay to go low, making the reset pulse width depend on only the delay of elements inside the pixel.

The sender's C-element circuit is shown in the lower-left corner of Figure 4. It has a flip-flop whose output (Apix) drives the column or row select line. This flip-flop is set when Aarb goes high; which happens when the arbiter selects that particular row or column. The flip-flop is reset when Rpix goes high, which happens when two conditions are satisfied: (i) Aarb is low, and (ii) all the pixels tied to the wired-OR line Rpix are reset. Thus the wired-OR serves three functions in this circuit: (i) It detects when there is a request in that row or column, passing on the request to the arbiter by taking Rarb high; (ii) it detects when the receiver acknowledges, by watching for Aarb to go low; and (iii) it detects when the pixel(s) in its row or column are reset.

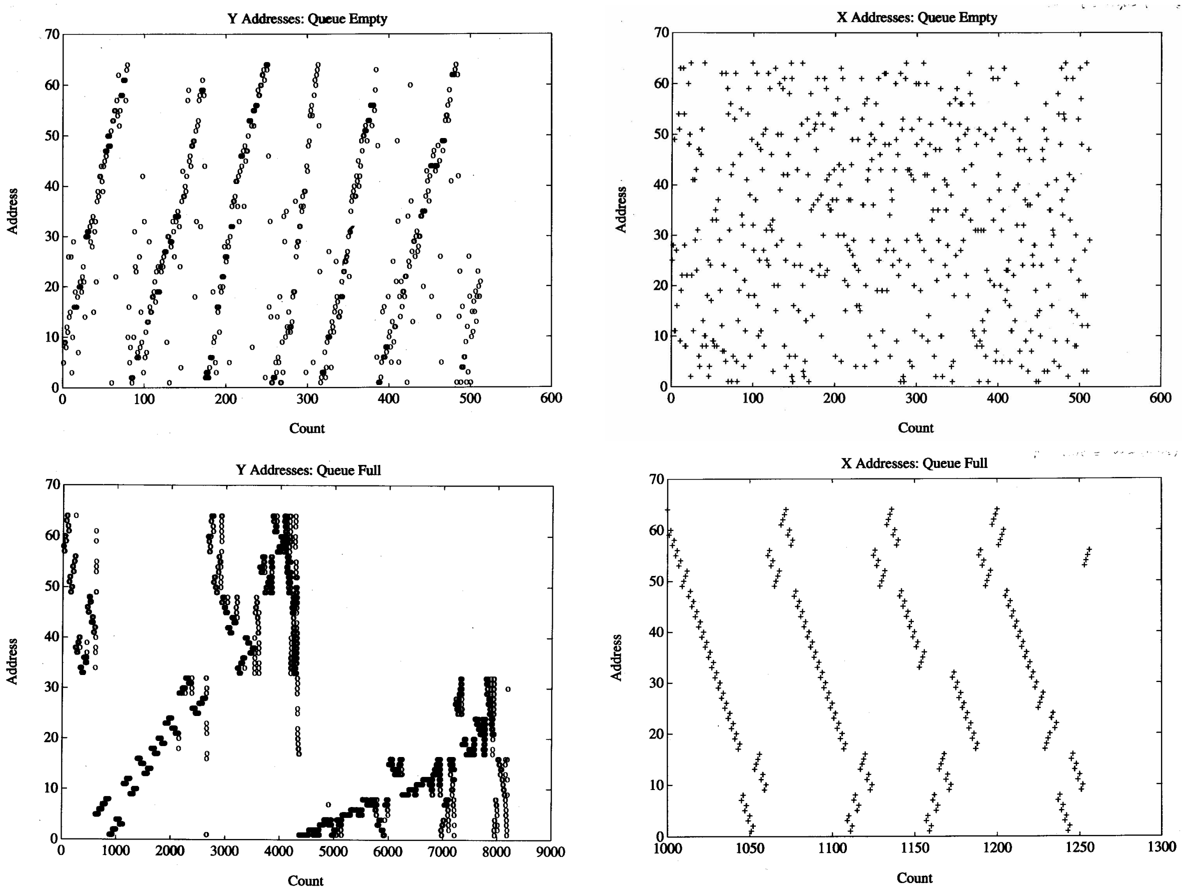


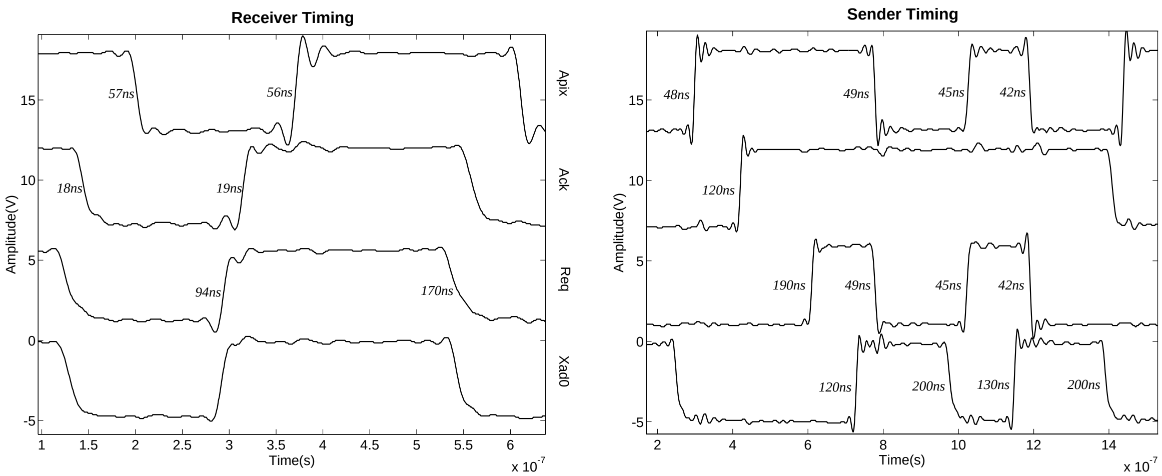
Fig. 6. Recorded address-event streams for small (queue empty) and large (queue full) channel loads. X and Y addresses are plotted on the vertical axes, and their position in the sequence is plotted on the horizontal axes. Queue Empty (top row): The Y addresses tend to increase with sequence number, but the row arbiter sometimes remains at the same row, servicing up to 4 neurons, and sometimes picks up a row that is far away from the previous row. Whereas the X addresses are distributed randomly. Queue Full (bottom row): The Y addresses tend to be concentrated in the top or bottom half of the range, or in the third quarter, and so on, and the column arbiter services all 64 neurons in the selected row. The X addresses tend to decrease with sequence number, as all the neurons are serviced progressively, except for transpositions that occur over regions whose width equals a power of 2. Note that the horizontal scale has been magnified by a factor of 30 for the X addresses.

There are two differences between my handshaking circuit and the handshaking circuit of Lazzaro et.al. [16].

First, Lazzaro et.al. disable all the arbiter's inputs to prevent it from granting another request while Ack is high. By using the AND gate at the top of the arbiter tree to disable the arbiter's *outputs* (Aarb) when Ack is high, my design leaves the arbiter's inputs undisturbed. As I explained in the previous subsection, my approach enables us to exploit locality in the arbiter tree and in the array.

Second, Lazzaro et.al. assume that the selected pixel will withdraw its request before the receiver acknowledges. This timing assumption may not hold if the receiver is pipelined. When the assumption fails, the row or column select lines may be cleared before the pixel has been reset. In my circuit, the row and column select signals (Apix) are reset only if Aarb is low, indicating that the receiver has acknowledged, *and* no current is being drawn from Rpix by the array, indicating that the pixel has been reset.

Mahowald's original design used a similar trick to ensure that the select lines were not cleared



(a) (b)

Fig. 7. Measured address-event channel timing. All the delays given are measured from the preceding transition: (a) Timing of Req and Ack signals relative to X-address bit (Xad0) and receiver pixel's acknowledge (Apix). Pipelining shaves a total of 113ns off the cycle time (twice the duration between Ack and Apix). (b) Timing of Req signal relative to the select signals fed into the top of the arbiter trees (Ysel and Xsel, disabled when Ack is high), and the Y-address bit (Yad0). Arbitration occurs in both the Y and X dimensions during the first cycle, but only in the X dimension during the second cycle; the cycle time is 730 ns for the first cycle and 420 ns for the second.

prematurely. However, her handshaking circuit used dynamic state-holding elements which were susceptible to charge-pumping and to leakage currents due to minority carriers injected into the substrate when devices are switched off. My design uses fully static stateholding elements.

The receiver's C-element (it is slightly different from the sender's) is shown in the lower-right corner of Figure 4. The C-element's output signal drives the chip acknowledge (Ack), strobes the latches, and activates the address decoders. The flip-flop that determines the state of Ack is set if Req is high, indicating that there is a request, *and* Apix is low, indicating that the decoders' outputs and the wired-OR outputs have been cleared. It is reset if Req is low, indicating that the sender has read the acknowledge, *and* Apix is high, indicating that the receiving neuron got the spike.

The address-bit latch and the logic inside the receiving pixel are also shown in the figure (middle of lower row and upper-right corner, respectively). The latch is opaque when Ack is high, and is transparent when Ack is low. The pixel logic produces an active low spike whose duration depends on the delay of the wired-OR and the decoder, and on the duration of the sender's reset phase.

Circuits for the blocks that are not described here—namely, the address encoder and the address decoder—are given in [4], [16].

4.4. Performance and Improvements

In this subsection, I characterize the behavior of the channel and present measurements of cycle times and the results of a timing analysis in this section. For unacessible elements, I have calculated estimates for the delays using the device and capacitance parameters supplied by MOSIS for the fabrication process.

Figure 6 shows plots of address-event streams that where read out from the sender under two vastly different conditions. In one case, the load was less than 5% of the channel capacity. In the other case, the load exceeded the channel capacity. For small loads, the row arbiter rearranges the address-events as it attempts to scan through the rows, going to the nearest row that is active. Scanning behavior is not evident in the X addresses because no more than 3 or 4 neurons are active simultaneously within the same row. For large loads, the row arbiter concentrates on one half, or one quarter, of the rows, as it attempts to keep up with the data rate. And the column arbiter services all the pixels in the selected row, scanning across the row. Sometimes the addresses are transposed because each arbiter cell chooses randomly between its left branch (lower half of its range) or its right branch (upper half of its range).

Figure 7a shows the relative timing of Req, Ack, the X-address bit Xad0, and the acknowledge from the pixel that received the address-event Apix. The cycle breaks down as follows. The 18ns delay between Req $\downarrow$ and Ack $\downarrow$ is due to the receiver's C-element (8.9ns) and the pad (9.1ns); the same holds for the delay between Req $\uparrow$ and Ack $\uparrow$. The 57ns delay between Ack $\downarrow$ and Apix $\downarrow$ is due to the decoder (13ns), the row wired-OR (41ns), and the second wired-OR that runs up the left edge of the array (3.4ns). The 57ns delay between Ack $\uparrow$ and Apix $\uparrow$, breaks down similarly: decoder (38ns), row wired-OR (14ns), left-edge wired-OR (3.4ns). The 94ns and 170ns delays between Apix and Req are due to the sender. The X-address bits need not be delayed because the C-element's 8.9ns delay is sufficient set-up time for the latches.

Figure 7b shows the relative timing of Req, the Y-address bits Yad0, and the select-enable signals (Xsel, Ysel) fed in at the top of the arbiter trees; Xsel and Ysel are disabled by Ack. The first half of the first cycle breaks down as follows. The 48ns delay between Req $\downarrow$ and Ysel $\uparrow$ is due to the receiver (18ns) and the AND circuit (30ns)—it is a chain of two 74HC04 inverters and a 74HC02 NOR gate.[3] The 120ns delay between Ysel $\uparrow$ and Yadr0 $\uparrow$ is due to the arbiter (propagating a high-going select down the six-level tree takes 19ns per stage and 6.7ns for the bottom stage), the row/column C-element (5.2ns), the address encoder (3.2ns), and the pad (7ns). The same applies to the delay between Xsel $\uparrow$ and Req $\uparrow$. The 190ns delay between Yad0 $\uparrow$ and Xsel $\uparrow$ is due to the column wired-OR (120ns), the arbiter (propagating a high-going request up the six-level tree takes 4.8ns per stage and 11ns for the top stage), the pad (7ns), and the AND circuit (30ns). The slow events are propagating a high-going select down the arbiter (100ns total) and propagating a request from the pixel through the column wired-OR (120ns).

The second half of the first cycle breaks down as follows. The 49ns delay between Req $\uparrow$ and Xsel $\downarrow$, Ysel $\downarrow$ is identical to that for the opposite transitions. The 200ns delay between Xsel $\downarrow$, Ysel $\downarrow$ and Req $\downarrow$ is due the arbiter (propagating a low-going select signal down the six-level tree takes 6.8ns per stage and 2.5ns for the bottom stage), the column and row wired-ORs (120ns), the handshake circuit (5.2ns), the encoder (32ns), and the pad (7ns). The slow events are propagating a low-going select down the arbiter (36ns total), restoring the wired-OR line (120ns), and restoring the address-lines (32ns).

The second cycle is identical to the first one except that there is no row arbitration. Hence, Xsel goes high immediately after Req goes low, eliminating the 310ns it takes to propagate a select signal down the Y-arbiter, to select a row, to get the column request from the pixel, and to propagate a request up the X-arbiter.

This channel design achieves a peak throughput of 2.5MS/s (million spikes per second), for a 64×64 array in 2μm CMOS technology. The cycle time is 730ns if arbitration is performed in both dimensions and 420ns when arbitration is performed in only the X dimension (i.e., the pixel sent is from the same row as was the previous pixel). These cycle times represent a threefold to fivefold improvement over the 2μs cycle times reported in the original work [4], and are comparable to the shortest cycle time of 500ns reported for a much smaller 10×10 nonarbitered array fabricated in 1.6μm technology [12]. Lazzaro et.al. report cycle times in the 100-140ns range for their arbitered design, but the array size and the chip size are a lot smaller.

Pipelining the receiver shaves a total of 113ns off the cycle time—the time saved by latching the address-event, instead of waiting for the receiving pixel to acknowledge. Pipelining the sending pixel's reset phase did not make much difference because most of the time is spent waiting for the row and column wired-OR request lines to reset, once the pixel itself is reset. Unfortunately, I did not pipeline reseting these request lines: I wait until the receiver's acknowledge disables the select signals that propagate down the arbiter tree, and Arb goes low, before releasing the column and row wired-OR request line. Propagating these high-going and low-going select signals down the six-level arbiter tree ($110\text{ns} + 40\text{ns}$) and resetting the column and row request lines (120ns) adds a total of 270ns to the cycle time, when arbitration is performed in only the X dimension, and adds 400ns when arbitration is performed in both the X and Y dimensions.

The imapct of the critical paths revealed by my timing analysis can be reduced significantly by making three architectural modifications to the sender chip:

1. Moving the AND gate that disables the arbiter's select signals to the bottom of the tree would shave off a total of 144ns; this change requires an AND gate for each row and each column.[4]
2. Removing the input to the column and row wired-OR from the arbiter would allow the column and row request lines to be cleared as soon as the pixel is reset; this requires adding some logic to reset the flip-flop in the sender's C-elemet when Rpix is high and Aarb is low.
3. Doubling the drive of the two series devices in the pixel that pull down the column line would reduce the delay of the wired-OR gate to 60ns.

These modifications, taken together, allow us to hide reseting the column and row select lines in the 59ns it takes for Apix$\uparrow \Rightarrow$ Req$\uparrow \Rightarrow$ Ack$\uparrow \Rightarrow$ Xsel$\downarrow$, shaving off a total of 120ns, when arbitration occurs in only the X dimension, and 180ns when arbitration occurs in both dimensions. These changes, together, will reduce the cycle time to 156ns with arbitration in one dimension, and to 406ns with arbitration in both dimensions. Further gains may be made by optimizing the sizes of the devices in the arbiter for speed and adding some buffers where necessary.

4.5. *Bugs and Fixes*

In this section, I describe the bugs I discovered in the channel design and propose some ways to fix them.

During testing, I found that the sender occasionally generates illegitimate addresses, i.e. outside the 1 to 64 range of pixel locations. In particular, row (Y) addresses higher than 64 were observed. This occurs when row 64 and one other row, or more, are selected simultaneously. I traced this problem to the sender's C-element (lower-left of Figure 4).

After Ack goes high and Aarb goes low, the pull-up starts charging up Rpix. When Rpix crosses the threshold of the inverter, Rarb goes low. When the arbiter sees Rarb go low it selects another row. However, if Rpix has not crossed the threshold for resetting the flip-flop, the flip-flop remains set and keeps the previous row selected. Hence, two rows will be selected at the same time, and the encoder will OR their addresses together.

This scenario is plausible because the threshold of the inverter that drives Rarb is lower than that of the flip-flop's reset input; I calculated 2.83V and 3.27V, respectively. If any neuron in the previously selected row spikes while Rpix is between these two values, Rpix will be pulled back low, and the flip-flop will not be reset. Rpix spends about $0.44/2.5\times120\text{ns} = 21\text{ns}$ in this critical window. At a total spike rate of 100KHz, we expect a collision rate of 0.05Hz, just for row sixty-four alone. I observed a rate of 0.06Hz; the higher rate observed may be due to correlations in firing times. To eliminate these collisions, we should disable neurons from firing while their row is selected (i.e., Ay is high). That way, Rpix will remain low until Apix goes low, ensuring that the flip-flop is reset.[5]

I also had to redesign the receiver pixel to eliminate charge-pumping and capacitive turn-on, which plagued the first pixel I designed, and to reduce cross-talk between the digital and analog parts by careful layout. Two generations of receiver pixel circuit designs are shown in Figure 8.

The pair of transistors controlled by the row and column select lines, Ry and Rx, pump charge to ground when non-overlapping pulses occur on the select lines. In my first design, this charge-pump could supply current directly to the integrators—irrespective of whether or not that pixel had been selected. The pump currents are significant as, on average, a pixel's row or column is selected 64 times more often than the pixel itself. For a parasitic capacitance of 20fF on the node between the transistors, an average spike rate of 100Hz per pixel, and a voltage drop of 0.5V, the current is 64pA. This current swamps out the subpicoamp current levels we must maintain in the diode-capacitor integrator to obtain time con-

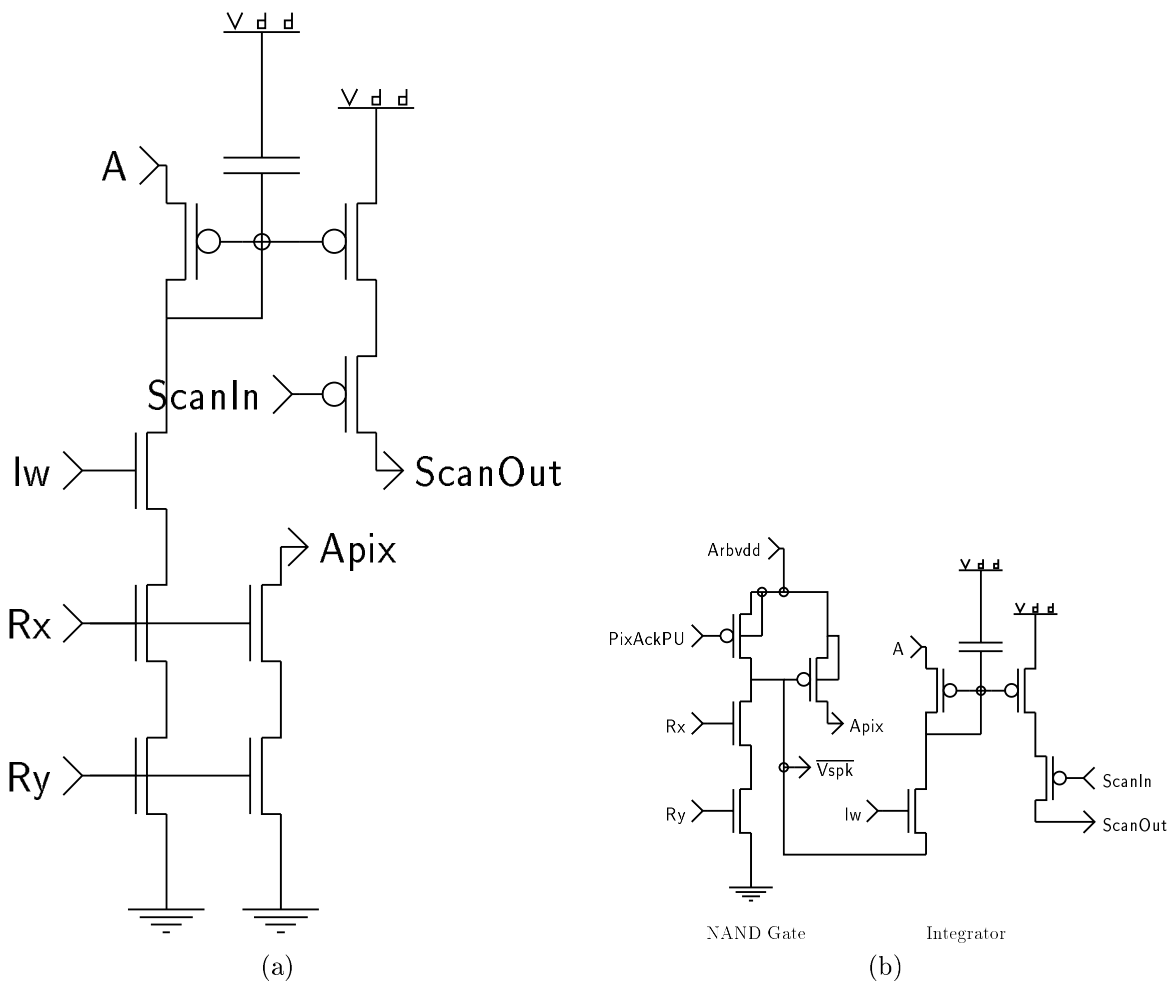


Fig. 8. Comparison between my (a) first and (b) second circuit designs for a receiver pixel. The second design eliminated charge-pumping and capacitive turn-on which plagued the first design, as explained in the text.

stants greater than 10ms using a tiny 300fF capacitor.

I solved this problem in my second design by adding a pull-up to implement an nMOS-style NAND gate. The pull-up can supply a fraction of a milliamp, easily overwhelming the pump current. The NAND gate turns on the transistor that supplies current to the integrator by swinging its source terminal from Vdd to GND. As demonstrated by Cauwenberghs [19], this technique can meter very minute quantities of charge onto the capacitor. In addition to eliminating charge-pumping, this technique circumvents another problem we encounter when we attempt to switch a current source on and off: capacitive turn-on.

Rapid voltage swings on the select line are transmitted to the source terminal of the current-source transistor by the gate-drain overlap capacitor of the switching transistor. In the first design, where this terminal's voltage was close to GND, these transients could drive the source terminal a few tenths of a volt below GND. As a result, the current source would pass a fraction of a picoamp even when Iw was tied to GND. In the new design, the pull-up holds this node up at Vdd and supplies the capacitive current, preventing the node from being discharged.

5. Discussion

Since technological limitations precluded the use of dedicated lines, I developed a time-multiplexed channel that communicates neuronal ensembles between chips, taking advantage of the fact that the bandwidth of a metal wire is several orders of magnitude greater than that of a nerve axon. Thus, we can reduce the number of wires by sharing wires among neurons. We replaced thousands of dedicated lines with a handfull of wires and thousands of switches (transistors). This approach paid off well because transistors take up much less real estate on the chip than wires do.

I presented three compelling reasons to provide random access to the shared channel, using event-driven communication, and to resolve contention by arbitration, providing a queue where pixels wait their turn. These choices are based on the assumption that activity in neuromorphic systems is clustered in time and in space.

First, unlike sequential polling, which rigidly allocates a fixed fraction of the channel capacity to each quantizer, an event-driven channel does not service inactive quantizers. Instead, it dynamically reallocates the channel capacity to active quantizers and allows them to samples more frequently. Despite the fact that random access comes at the cost of using $\log_2 N$ wires to transmit addresses, instead of just one wire to indicate whether a polled node is active or not, the event-driven approach results in a lower bit rate and a much higher peak sampling rate when activity is sparse.

Second, an arbiterless channel achieves a maximum throughput of only 18% of the channel capacity, with an extremely high collision rate of 64 percent. Whereas an arbitered channel can operate at 95% capacity without any losses due to collisions—but its latency and temporal dispersion is 10 times the cycle period. Thus, unless the cycle-time of the arbiterless channel is 5 times shorter, the arbitered channel will offer higher performance in terms of the number of spikes that get through per second. Furthermore, the cycle-time of the arbiterless channel must be even shorter if low error rates are desired, as failure probabilities of 5 percent require it to operate at only 2.5 percent of its capacity. A comparable error in timing precision due to temporal dispersion in the arbitered channel occurs at 84.8% of the channel capacity, using the numbers given in Section 3.3.

And third, the arbitered channel scales much better than the nonarbitered one as the technology goes to finer feature sizes, yielding higher levels of integration and faster operation. As the number of neurons grows, the cycle time must decrease proportionately in order to obtain the desired throughput. Hence, there is a proportionate decrease in queueing time and in temporal dispersion—even though the number of cycles spent queueing remains the unchanged when the same fraction of the channel capacity is in use. Whereas the collision probability remains unchanged under the same conditions.

I described the design and operation of an event-driven, arbitered interchip communication channel that reads out pulse trains from a 64×64 array of neurons on one chip and transmits them to corresponding locations on a 64×64 array of neurons on a second chip. This design acheived a threefold to fivefold improvement over the first-generation design [4] by introducing three new enhancements.

First, the channel used a three-stage pipeline, which allowed up to three address-events to be processed concurrently. Second, the channel exploited locality in the arbiter tree by picking the input that was closest to the previously selected input—spanning the smallest number of levels in the tree. And third, the channel exploited locality in the row-column organization by sending all requests in the selected row without redoing the arbitration between columns.

I identified three inefficiencies and one bug in my implementation, and I suggested circuit modifications to address these issues. First, to reduce the propagation delay of the acknowledge signal, we must remove the AND gate at the top of the arbiter tree, and disable the select signals at the bottom of the arbiter tree instead. Second, to reset the column and row wired-OR lines while the receiver is latching the address-event and activating the acknowledge signal, we must remove the input to the wired-OR from the arbiter and redesign the row/column C-element. Third, to decrease the time the selected row takes to drive its requests out on the column lines, we must double the size of the pull-down devices in the pixel. And fourth, to fix the multiple-row–selection bug, we must guarantee that the row request signal is stable by disabling all the neurons in a row whenever that row is selected.

These modifications will provide error-free operation and will push the capacity up two and a half times, to 6.4MS/s. According to my calculations, for neurons with a mean latency of 2ms, a coefficient of variation of 10%, and an firing-rate attenuation factor of 40, this capacity will be enough to service a population of up to 82,000 neurons.

However, as the number of neurons increases, the time it takes for the pixels in the selected row to drive the column lines increases proportionately. As this interval is a significant fraction of present design's cycle time (38% in the modified design), the desired scaling will not be acheived unless the ratio between the unit current and the unit capacitance increases linearly with integration density. SRAM and DRAM scaling trends indicate that this ratio increases sublinearly, and hence the present architecture will not scale well. We need to develop new communication channel architectures to address this issue.

Acknowledgements

This work was partially supported by the Office of Naval Research; DARPA; the Beckman Foundation; the Center for Neuromorphic Systems Engineering, as part of the National Sceince Foundation Engineering Research Center Program; and the California Trade and Commerce Agency, Office of Strategic Technology.

I thank my thesis advisor, Carver Mead, for sharing his insights into the operation of the nervous system. I also thank Misha Mahowald for making available layouts of the arbiter, the address encoders, and the address decoders; John Lazzaro, Alain Martin, Jose Tierno, and Tor (Bassen) Lande for helpful discussions on address events and asynchronous VLSI; Tobi Delbrück for help with the Macintosh address-event interface; and Jeff Dickson for help with PCB design.

Notes

1. This result is derived by assuming independent firing probabilities and approximating the resulting binomial distribution with the Poisson distribution.
2. This result is also based on the assumption that samples arrive according to a Poisson process.
3. Xsel does not go high at this point because the X-arbiter has not received any requests, as a row has not yet been selected.
4. This modification was suggested to me by Tobi Delbrück.
5. This solution was suggested to me by Jose Tierno.

References

1. M Schwartz. *Telecommunication Networks: Protocols, Modeling, and Analysis.* Addison-Wesley, Reading, MA, 1987.
2. A S Tanenbaum. *Computer Networks.* Prentice-Hall International, 2 edition, 1989.
3. A Murray and L Tarassenko. *Analogue Neural VLSI: A Pulse Stream Approach.* Chapman and Hall, London, England, 1994.
4. M Mahowald. *An Analog VLSI Stereoscopic Vision System.* Kluwer Academic Pub., Boston, MA, 1994.
5. M Sivilotti. *Wiring considerations in Analog VLSI Systems, with application to Field-Programmable Networks.* PhD thesis, California Institute of Technology, Pasadena CA, 1991.
6. C A Mead. Neuromorphic electronic systems. *Proc. IEEE*, 78(10):1629–1636, 1990.
7. K A Boahen. Retinomorphic vision systems. In *Microneuro'96: Fifth Intl. Conf. Neural Networks and Fuzzy Systems*, pages 2–14, Los Alamitos CA, Feb 1996. EPFL/CSEM/IEEE, IEEE Comp. Soc. Press.
8. K A Boahen. Retinomorphic vision systems ii: Communication channel design. In *Proc. IEEE Intl. Symp. Circ. and Sys.*, volume Supplement, Piscataway NJ, May 1996. IEEE Circ. & Sys. Soc., IEEE Press.
9. K A Boahen. A retinomorphic vision system. *IEEE Micro*, 16(5):30–39, October 1996.
10. K A Boahen. The retinomnorphic approach: Pixel-parallel adaptive amplification, filtering, and quantization. *Analog Integr. Circ. and Sig. Proc.*, 13:53–68, 1997.
11. K A Boahen. *Retinomorphic Vision Systems: Reverse Engineering the Vertebrate Retina.* PhD thesis, California Institute of Technology, Pasadena CA, 1997.
12. A Mortara, E Vittoz, and P Venier. A communication scheme for analog vlsi perceptive systems. *IEEE Trans. Solid-State Circ.*, 30(6):660–669, 1995.
13. A Abusland, T S Lande, and M Hovin. A vlsi communication architecture for stochastically pulse-encoded analog signals. In *Proc. IEEE Intl. Symp. Circ. and Sys.*, volume III, pages 401–404, Piscataway NJ, May 1996. IEEE Circ. & Sys. Soc., IEEE Press.
14. L Kleinrock. *Queueing Systems.* Wiley, New York NY, 1976.
15. I E Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

16. J Lazzaro, J Wawrzynek, M Mahowald, M Sivilotti, and D Gillespie. Silicon auditory processors as computer peripherals. *IEEE Trans. on Neural Networks*, 4(3):523–528, 1993.
17. J Lazzaro, J Wawrzynek, and A Kramer. Systems technologies for silicon auditory models. *IEEE Micro*, 14(3):7–15, 1994.
18. K A Boahen. Retinomorphic vision systems i: Pixel design. In *Proc. IEEE Intl. Symp. Circ. and Sys.*, volume Supplement, pages 9–13, Piscataway NJ, May 1996. IEEE Circ. & Sys. Soc., IEEE Press.
19. G Cauwenberghs. A micropower cmos algorithmic a/d/a converter. *IEEE Transactions on Circuits and Systems I*, 42(11), 1995.

Kwabena A. Boahen is an Assistant Professor in the Bioengineering Department at the University of Pennsylvania, Philadelphia PA. He received a PhD in Computation and Neural Systems from the California Institute of Technology, Pasadena, CA. He earned BS and MSE degrees in Electrical Engineering from the Johns Hopkins University, Baltimore MD, in the concurrent masters–bachelors program. His current research interests include mixed-mode multichip VLSI models of biological sensory systems, and asynchronous digital interfaces for interchip connectivity.

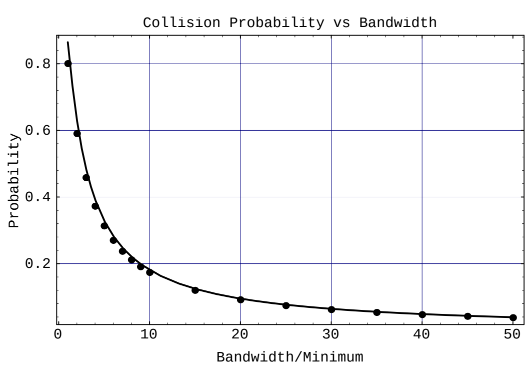


Fig. 9. Theoretical collision probability versus channel capacity (bandwidth) for Gaussian bursts. Actual capacity is normalized to the minimum capacity requirement. Thus, the numbers on the Bandwidth/Minimum axes are the reciprocal of the offered channel load G. For collision probabilities below 18 percent, capacities ten or more times the minimum are required. The dots represent results of a numerical computation based on the Gaussian and binomial distributions for sample occurrence within a burst, and the number of samples in each time-slot, respectively. The line represents results of a simplified, analytically tractable, approximation based on the equivalent rectangular and Poisson distributions, respectively. The simplified model overestimates the collision probability by no more than 8%; the estimation error drops below 4.4% for capacities greater than 10 times the theoretical minimum.

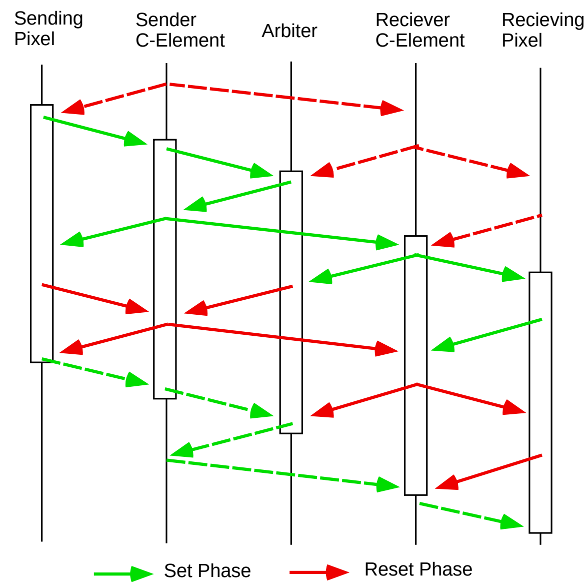


Fig. 10. Pipelined communication cycle sequence for arbitration in one dimension, showing four-phase minicycles among five elements. The boxes indicate the duration of the current cycle, which may overlap with the reset phase of the preceding cycle and the set phase of the succeeding cycle. (Steps associated with the preceding and succeeding cycles are shown with dashed-lines.) Thus three address-events may be at different stages in the communication channel at the same time. The cycle consists of three smaller interwoven minicycles: sending pixel to C-element, C-element to C-element, and C-element to receiving pixel. The C-elements—also known as handshake circuits—ensure that the minicycles occur in lock-step, synchronizing the activity of the sending pixel, the arbiter, and the receiving pixel.