A Throughput-On-Demand Address-Event Transmitter for Neuromorphic Chips

Kwabena Boahen
University of Pennsylvania
Dept of Bioengineering
3320 Smith Walk
Philadelphia PA 19104-6392
kwabena@neuroengineering.upenn.edu

Abstract

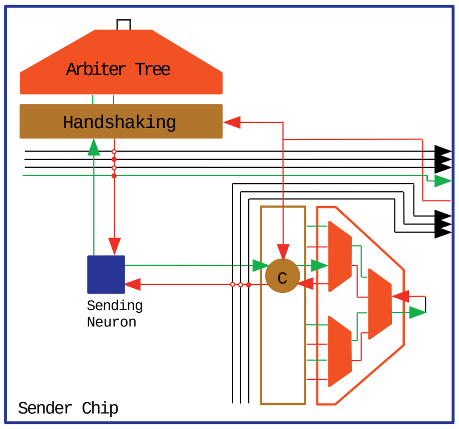
I present a scalable 2-D address-event transmitter interface designed to take advantage of the high integration densities available with advanced submicron technology. To sustain throughput, it exploits the linear increase in the number of active neurons per row with array size, instead of counting on a linear increase in the unit-current/unit-capacitance ratio, as existing designs do. I synthesize an asynchronous implementation starting from a high-level specification, and present test results from a 104×96*-neuron chip fabricated in a 1.2μm CMOS process. Reading out the state of all neurons in a selected row in parallel, and sending their spikes in a tight burst of events, yields cycle times between 40 to 70ns—six to ten times shorter than the 420ns minimum cycle time reported in earlier work.*

1: Communication channel scaling for 2-D arrays

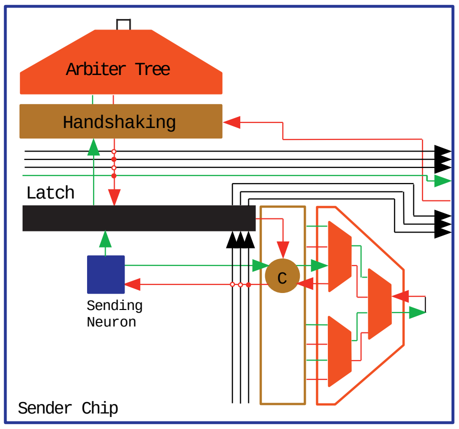
Random-access, time-multiplexed, communication channels have been developed to communicate spike trains between arrays of silicon neurons on multiple chips [11, 7, 12]. These communication channels find application in retinomorphic imager chips [7, 1, 4], and silicon auditory preprocessors [5], which perform pixel-parallel adaptive quantization. Several ways of maximizing and allocating channel capacity have been investigated, including polling (i.e., scanning) [11], free-for-all [10], and arbitration [12, 5, 7], and the merits of each approach have been debated [2]. However, little attention has been paid to the scaling properties of the architectures proposed.

For two-dimensional (2-D) architectures, where neurons are organized into rows and columns, cycle time is proportional to array size, due to increasing row and column line capacitance. Migrating to a more advanced fabrication process does not provide relief—unless the unit-current/unit-capacitance ratio is proportional to the integration density. Unfortunately, this current-drive ratio increases sublinearly with integration density, as evidenced by the scaling trends of DRAM and SRAM. Hence, existing 2-D communication channel architectures cannot take full advantage of the higher levels of integration offered by advanced submicron technologies.

In this paper, I describe an interchip communication channel for 2-D neuromorphic arrays that achieves improved scaling behavior by reading spikes from the array in parallel. When the current-drive ratio is constant, its throughput is sustained as the array size increases. And when the current-drive ratio increases, its throughput *improves* as the



(a)



(b)

Figure 1: Throughput-on-Demand Architecture
(a) Silicon neurons convert analog current into spikes (i.e., digital pulses). A neuron interface circuit communicates spikes to circuitry on the periphery of the array using a pair of request/select lines. A row/column interface circuit relays requests from a row or column of neurons to the arbiter and relays acknowledges from the arbiter back; it also activates the address encoder. Row/column arbitration is hierarchical: A neuron first sends a request to the row arbiter. When its row is selected, it sends a request to the column arbiter. When both its row and its column are selected, it clears its spike and withdraws its requests. (b) Throughput is boosted by reading spikes in parallel—using one line per column—from the selected row, storing them in a latch, and selecting the next row while they are transmitted. The row's address is also stored in the latch. The latch sends an acknowledge signal back to the row-interface circuits; it makes sure the column lines are cleared before withdrawing this signal.

feature sizes shrink. This behavior is achieved by reading out the states of all neurons in a selected row in parallel over the column lines. My new architecture is contrasted with the prior art in Figure 1.

The amount of parallelism achieved increases with the level of activity (i.e., what fraction of a row's neurons are active, on average, each time it is selected). For a given level of activity, the number of active neurons in the row increases as the array size increases. Hence, the number of spikes read out in parallel scales with the time required to drive the column lines. And, therefore, the channel throughput is sustainable. Furthermore, as the activity level goes up, this fraction goes up because the probability, per unit time, of a neuron spiking is higher and more time is spent servicing each row. Thus, more parallelism is achieved, and throughput goes up as the load increases.

I describe the design and performance of this throughput-on-demand communication channel in the following sections.

2: Concurrent hardware processes

Our goal is to design a communication channel that encodes activity on its N service ports as $\lceil \log_2(N) \rceil$-bit words on a single output port. Starting with a high-level specification, I implement the channel by performing a series of program decompositions. There are two advantages of reducing the channel process into concurrent subprocesses:

- We can use a divide-and-conquer strategy to reduce the logical complexity.
- We can share expensive hardware resources.

However, for this approach to be successful, we must synchronize the activities of these concurrent processes and resolve contention for shared resources. To this end, I follow a correct-by-construction synthesis methodology for asynchronous digital VLSI systems developed by Martin [8].

The address-event transmitter is defined in the concurrent–hardware-processes (CHP) description language [8] as follows:

$$\begin{aligned} \mathsf{AER}(N) \quad \equiv \quad & \underline{\mathsf{process}}(\langle, n : 1..N : L_n\rangle, A!\mathsf{int}(\lceil \log_2(N)\rceil)) \\ & *[[\langle | \; n : 1..N : \overline{L_n} \rightarrow A!\mathsf{enc}(n) \| L_n\rangle]] \\ & \underline{\mathsf{end}} \end{aligned}$$

Thus, the interface has N dataless ports, L_n, and a single output port, A, that transmits a $\lceil \log_2(N)\rceil$-bit word. These bits encode the identity of the active input port chosen by arbitration. The function $\mathsf{enc}(n)$ converts an N-bit one-hot representation into a $\lceil \log_2(N)\rceil$-bit binary representation, where n is the position of the bit that is true in the N-bit word.

We realize three optimizations by reducing this high-level specification into a hierarchy of concurrent processes:

1. We reduce the number of arbiters cells and encoder cells from $Y \times X$ to $Y + X - 2$ and $Y + X$, respectively, by going to a Y-row/X-column organization, where $Y \times X \equiv N$.
2. We boost throughput by performing arbitered sequential transmission of spikes from a previously selected row concurrently with parallel readout of spikes from a newly selected row.
3. We reduce the number of column lines by X—and eliminate the need to detect completion of concurrent column communications inside the array—by using a straight-data encoding.

The first optimization was introduced in Mahowald and Sivilotti's pioneering work [12, 7, 5]. The second and third optimizations—which exploit the first one—are innovations introduced in the present work; they are described in detail in this section. Prior to this work, I reduced the 2μs cycle-time of the first-generation design [7] to 420ns by exploiting locality in the array (i.e., servicing all requests in a chosen row before picking another row) and in the arbiter (i.e., spanning the smallest subtree that includes another request) [2].

2.1: Reorganizing into rows and columns

We reorganize the N service ports into rows and columns to reduce the number of arbiters and encoders from $\mathsf{O}(N)$ to $\mathsf{O}(N^{1/2})$. We start by introducing hierarchical row/column arbitration:

$$\mathsf{AER}(Y, X) \equiv *[[\langle | \; y : 1..Y : \langle \vee x : 1..X : \overline{L_{y \cdot x}}\rangle \rightarrow [\langle | \; x : 1..X : \overline{L_{y \cdot x}} \rightarrow A!\mathsf{enc}(y \cdot x) \| L_{y \cdot x}\rangle]\rangle]]$$

where $Y \times X = N$ and $y \cdot x \equiv Xy + x$. Observe that, first we use a Y-input arbiter to choose one of Y rows, and then we use a X-input arbiter to choose one of X ports assigned to that row. Hierarchical arbitration guarantees that only one row is active at any given instant. Hence, we can share a single X-input column-arbiter between all the rows. We must OR together all requests within each row to generate requests for the row-arbiter. Replacing $XY - (Y + X - 2)$ arbiters with $X \times Y$ OR gates is a big win, as arbitration is a lot more expensive that ORing. A similar reduction in the number of encoder cells is also achieved.

We can save time by servicing all requests in a chosen row before we pick another row [2]. Our goal, then, is to allow all requests in the selected row to compete in parallel for access to the column arbiter. It is realized by rewriting the array process:

```
$\mathsf{ARRAY}(Y, X) \equiv$
$\underline{\mathsf{process}}(\langle , n : 1..YX : L_n\rangle, \langle , x : 1..X : C_x\rangle, \langle , y : 1..Y : R_y\rangle, \langle , x : 1..X : A_x\rangle, \langle , y : 1..Y : B_y\rangle)$
 $*[[\langle$     $\|y : 1..Y : \langle \vee x : 1..X : \overline{L_{y \cdot x}}\rangle \rightarrow$
           $R_y; A_y \| \langle \| x : 1..X : [\overline{L_{y \cdot x}} \rightarrow C_x; A_x \| L_{y \cdot x}; C_x \mathbin{[\!]} \neg\overline{L_{y \cdot x}} \rightarrow \mathsf{skip}]\rangle; R_y$
   $\rangle]]*$
$\underline{\mathsf{end}}$
```

$\mathsf{ARB}(N) \equiv *[[\langle | \; n : 1..N : \overline{L_n} \rightarrow L_n; L_n\rangle;]]$, $\mathsf{ENC}(N) \equiv *[[\langle \mathbin{[\!]} n : 1..N : \overline{L_n} \rightarrow A!\mathsf{enc}(n) \bullet L_n\rangle]]$

I extracted the arbiters and encoders, leaving a $Y \times X$ array of N service ports. Thus, $\mathsf{AER}(N)$ is now made up of five concurrent processes: The array ($\mathsf{ARRAY}(Y, X)$), an X-input column-arbiter ($\mathsf{ARB}(X)$), an X-input column-address–encoder ($\mathrm{ENC}(X)$), a Y-input row-arbiter ($\mathrm{ARB}(Y)$), and a Y-input row-address–encoder ($\mathrm{ENC}(Y)$). In addition to its XY service ports (L_n), $\mathsf{ARRAY}(Y, X)$ has two sets of Y ports (R_y and B_y) and two sets of X ports (C_x and A_x). It uses these ports to communicate with the row arbiter (R_y), the row-address encoder (B_y), the column arbiter (C_x), and the column-address encoder (A_x). The bullet operation ($\bullet$) in $\mathsf{ENC}(N)$ allows the next L_n communication to begin as soon as the last one has completed—even if the last A communication is not finished.[1]

When we OR requests from the same row to generate a request for the column arbiter, we should not wait for inactive ports to communicate on the column lines when the row is selected. The forced-choice selection between $\overline{L_{y \cdot x}}$ and the negated probe, $\neg\overline{L_{y \cdot x}}$, in ARRAY ensures this. Concurrent communication on the C_x ports is not prolonged by waiting for probes on inactive ports to become true. A problem that did not occur in the original program because, for a given row, we selected active ports one by one and communicated on them sequentially—and we knew that at least one probe would be true because we ORed all of them together. The forced-choice selection has an additional effect: It allows only neurons that were active *at the time that the row was selected* to participate. Thus, subsequent activity does not perturb the completion-detection process.

Our final program decomposition makes the array's row/column organization explicit by introducing separate row and column processes.

$$\begin{array}{lcl} \mathsf{ROW}(X) & \equiv & *[[\langle \vee x : 1..X : \overline{L_x}\rangle \rightarrow R; A \| \langle \| x : 1..X : [\overline{L_x} \rightarrow C_x \| L_x \mathbin{[\!]} \neg\overline{L_x} \rightarrow \mathsf{skip}]\rangle; R;]] \\ \mathsf{COL}(Y) & \equiv & *[[\langle \mathbin{[\!]} y : 1..Y : \overline{L_y} \rightarrow C; A \| L_y; C\rangle]] \end{array}$$

As shown in Figure 1a, each row process, $\mathrm{row}(y)$, probes the port $L_{y \cdot x} = \mathrm{row}(y).L_x$ to see if any of its neurons are active. If so, it makes a request to the row-arbiter on the channel $(\mathrm{row}(y).R, \mathrm{arby}.L_y)$. When its request is granted, it activates its X dataless ports, $(\mathrm{col}(x).L_y, \mathrm{row}(y).C_x)$, in parallel and tells the row-encoder, $(\mathrm{row}(y).A, \mathrm{adry}.L_y)$, to output its address. These communications are relayed to the column-arbiter by X column processes, $(\mathrm{col}(x).C, \mathrm{arbx}.L_x)$. When a column's request is granted, it completes its communication with the row-process and calls the column-encoder, $(\mathrm{col}(x).A, \mathrm{adrx}.L_x)$, which outputs its address. As these column communications are completed sequentially, the column addresses of active neurons in the selected row are transmitted one by one.

[1]By definition of the bullet, when one operation is finished, we know the other has started. Hence, there is no danger of performing several L_n communications without doing the corresponding A communications.

2.2: Reading the array in parallel

We can increase throughput by reading spikes from a row in parallel, combining its X dataless ports into a single X-bit data port. Thus, a row indicates concurrent activity on its X service ports by setting the corresponding bit in the X-bit word. Of course, we need to combine the X $\mathsf{COL}(Y)$ processes into a single X-bit wide column bus to match. The column communications may be merged as follows:

$$
\begin{array}{lcl}
\mathsf{ROW}(X) & \equiv & \underline{\mathsf{process}}(\langle, x : 1..X : L_x\rangle, C!\mathsf{int}(X), R, A) \\
 & & b : \mathsf{int}(X) \\
 & & *[\langle \| x : 1..X : [\overline{L_x} \rightarrow b.x\uparrow \;[\!]\; \neg\overline{L_x} \wedge \langle \vee i : 1..X : b.i\rangle \rightarrow \mathsf{skip}]\rangle; \\
 & & \quad R; A \| (C!b; \langle \| x : 1..X : [b_x \rightarrow L_x \| b_x\downarrow \;[\!]\; \neg b.x \rightarrow \mathsf{skip}]\rangle); R;] \\
 & & \underline{\mathsf{end}} \\
\mathsf{BUS}(Y, X) & \equiv & \underline{\mathsf{process}}(\langle, y : 1..Y : L_y?\mathsf{int}(X)\rangle, C!\mathsf{int}(X)) \\
 & & *[[\langle [\!] y : 1..Y : \overline{L_y} \rightarrow C!(L_y?)\rangle]] \\
 & & \underline{\mathsf{end}} \\
\mathsf{LATCH}(X) & \equiv & \underline{\mathsf{process}}(L?\mathsf{int}(X), \langle, x : 1..X : C_x\rangle, \langle, x : 1..X : A_x\rangle) \\
 & & b : \mathsf{int}(X) \\
 & & *[L?b; \langle \| x : 1..X : [b.x \rightarrow C_x; A_x \| b.x\downarrow; C_x \;[\!]\; \neg b.x \rightarrow \mathsf{skip}]\rangle;] \\
 & & \underline{\mathsf{end}}
\end{array}
$$

I have combined X $\mathsf{COL}(Y)$ processes into a single process called $\mathsf{BUS}(Y, X)$, added an X-bit state variable, b, to $\mathsf{ROW}(X)$, and added a X-bit latch process, $\mathsf{LATCH}(X)$. These processes are organized as shown in Figure 1b. I allow $\mathsf{ROW}(X)$'s service ports (i.e., L_x) to modify b's bits concurrently; writing is terminated when at least one bit is set. $\mathsf{ROW}(X)$ then makes a request to the row-arbiter (i.e., R). When it is selected, it outputs b onto the column bus (i.e., $C!b$). $\mathsf{BUS}(Y, X)$ transfers the data to the latch (i.e., $C!(L_y?)$). For every bit $b.x$ that is set, $\mathsf{LATCH}(X)$ communicates with the corresponding input, C_x, of the column-arbiter; these communications occur concurrently. When the arbiter responds, the address-encoder is asked to output the column's address (i.e., A_x) and the bit is reset. Hence, the bits are reset sequentially, as columns are selected one by one and their addresses are transmitted.

For high speed, we must allow the row's parallel write (i.e., $L?b$ in $\mathsf{LATCH}(X)$) to occur concurrently with the arbiter's serial reads (i.e., C_x in $\mathsf{LATCH}(X)$). To be more precise, the latch and the array should only be synchronized at the beginning of the sequence, when data is transferred. The reset of the data lines should be acknowledged asynchronously with on-going latch-to-arbiter communications. This way, new data shows up from the array, ready to be written, as soon as the serial readout is completed. Specifying the detailed relationship between particular communication-cycle phases is not possible at this level of abstraction, so we will wait till we get to a lower level to pursue this part of the design further.

Having decomposed the address-event transmitter single-line CHP programs—while optimizing the utilization of time and space—our next step is to expand each communiation into a four-phase handshake.

3: Implementing concurrent hardware processes

We now hand-compile our decomposed CHP programs into production rules, which can be implemented directly with CMOS transistors [8]. First, we perform handshaking expansion (HSE), which involves fleshing out each communication into a four-phase handshake on

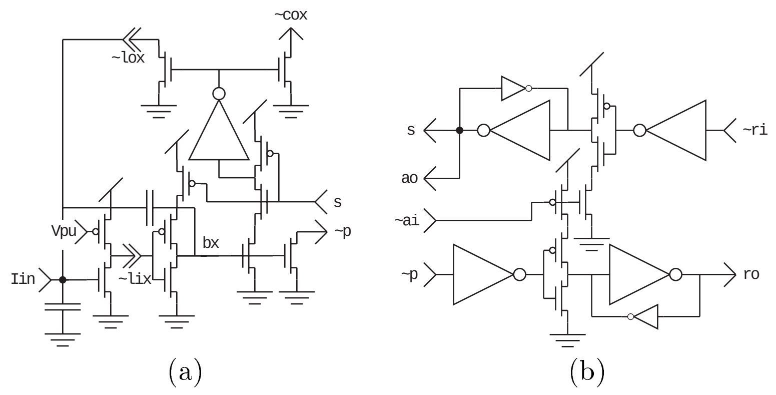


Figure 2: Neuron and Row/Column Interface Circuits
(a) This circuit interfaces the neuron (i.e., `l̃ix,l̃ox`) with the row interface circuit (i.e., `p̃,s`) and the latch cell (i.e., `c̃ox`). The pull-downs on `p̃` and `c̃ox` form row-wide and column-wide wired-NOR gates; a current source at the edge of the array serves as the pull-up. The neuron is disabled when the row is selected (i.e., `s` is high) to prevent generation of new spikes. The capacitive positive-feedback provides hysteresis [9] and speeds up the cycle, making `lix` change at close to 1V/ns—even when the neuron's input capacitor charges at only 1V/ms. And the current-starved nMOS-style inverter—which operates with less than a microamp of bias current—limits static power dissipation [6]. (b) This circuit interfaces either the row or the latch cell (i.e., `p̃,s`) with the arbiter (i.e., `ro, r̃i`) and the address-encoder (i.e., `ao, ãi`). It consist of two aC-elements—gates whose outputs are set when both inputs are high and cleared when a given input is low.

a pair of lines, assigning an active or passive role to each port, and choosing a data representation. For each subroutine, we realize two kinds of optimizations by reshuffling the sequence in which the four phases of each communication cycle occur—with respect to those of other communications in the same sequence.

- We can improve speed by performing the second half of the communication cycle concurrently with succeeding communications—in cases where it is not used for synchronization.
- We can reduce memory by carefully deciding when output transitions occur—co-opting them to store state information.

For each pair of ports that form a channel, we must choose which port is active and which is passive (i.e. which port initiates the communication). Of course, in cases where a port is probed, it must be passive [8]. For the other cases, we have complete flexibility, and we may seek the option that results in a more efficient solution.

3.1: Rowing along

We start by performing HSE on ROW(X), making L_x passive, R active, and C active. We ignore A for now and use a straight-data representation for C (i.e., a single line per bit and no separate request line). Thus, we obtain:[2]

[2]For compactness, I introduce a vector notation for HSE: Given `{x = 1..X}`, we expand `{ax *}` `= a1 * a2 * ... * aX`. There can be more than one dimension: Given `{x = 1..X}`,`{y = 1..Y}`, we expand `{{ayx *} @ ay:} = (a11 * a12 * ... * a1X) @ a1: (a21 * a22 * ... * a2X) @ a2: ... : (aY1 * aY2 * ... * aYX) @ aY:`—`x` is expanded first because it appears only at the first level.

```
# ({lix,},ri,ci)row({lox,},ro,{cox,}) {x = 1..X} #
*[{lix -> bx+,}; {bx |}; ro+; [ri]; {(bx -> cox+,lox+; ~lix -> bx-) ||};
[ci & {~bx &}]; {cox-,lox-,}; [~ci]; ro-; [~ri];]
```

I implemented the two R's with a pair of two-phase communications (i.e., two halves of a four-phase communication). Using a straight-data representation requires just X data lines, compared to $2X$ for a delay-insensitive dual-rail representation. The original unmerged column communications, with X dataless channels, also required $2X$ lines [12, 7, 5, 2]. Therefore, merging the column communications saves X column lines—and also allows us to push detecting completion of X concurrent column communications outside the array.

We can give the arbiter an early start on selecting the next row by moving the second R communication in front of the second half of C. Thus, we have

```
# ({lix,},ri,ci)row({lox,},ro,{cox,}) {x = 1..X} #
*[{lix -> bx+,}; [{bx |}]; ro+; [~ci & ri]; {(bx -> cox+,lox+;
  ~lix -> bx-) ||}; [ci & {~bx &}]; ro-; [~ri]; {cox-,lox-,};]
```

This reshuffling is dangerous, however, because the arbiter can select another row immediately after `[~ri]`—before the previous row releases the column lines. To guarantee mutual exclusion, I have moved `[~ci]` forward to where `[ri]` occurs in the next cycle, effectively blocking a newly selected row from proceeding until the ongoing column communication is completed. We also get more time to complete the column communication to boot. For the same reason, it is beneficial to delay `[ci]` as long as possible. However, we cannot move it past `ro-`, because `ci` must be high in order to lock out the other rows.

Now let us break the reshuffled sequence in two, the pixel part and the control part, and introduce two intermediaries, `p` and `s`, for them to communicate with.

```
  *[{lix -> bx+,}; {bx |} -> p+; [s]; {(bx -> cox+,lox+; ~lix -> bx-) ||};
    {~bx &} -> p-; {~s -> cox-,lox-,};]
||*[[p]; ro+; [ri & ~ci]; s+; [~p & ci]; ro-; [~ri]; s-;]
```

`p` signals when the row wants to make a request (it is set when any `bx` is set and cleared when all `bx` are cleared) and `s` signals when the row is selected.

Our final sequence operates as follows. When a spike occurs (i.e., `lix` becomes true), the bit is set (i.e., `bx+`) and the pixel drives the row-request line high (i.e., `p+`). The controller relays this request to the row arbiter (i.e., `ro+`) and acknowledges the pixel by driving the row-select line high (i.e., `s+`) when the arbiter acknowledges. If necessary, the controller waits until ongoing column communications are completed. When the row is selected, all pixels with spikes place requests on their column lines (i.e. `cox+`), clear their spikes (i.e., `lox+`), and clear their bits. When all the bits have been cleared, the row-request line goes low. The controller responds by withdrawing the request it made to the arbiter, but it waits till it receives an acknowledge from the column bus, since this signal prevents interference with ongoing communications. As soon as the arbiter clears its acknowledge, the controller deselects the row. The pixels then release the column lines and clear their service acknowledges.

We are yet to implement the $\neg\overline{L_x} \rightarrow \mathsf{skip}$ part of the $\overline{L_x}/\neg\overline{L_x}$ selection, which allows the second R communication to start as soon as any of the bits are set, and freezes the row's state. We can use the upward transition on `s` to lock-in the state of `bx`. Thus, we obtain the following PRS:

```
# row {x = 1..X} #          { bx |}    -> p+                p         -> ro+
{ lix (& ~s) -> bx+ }       {~bx &}    -> p-               ~p  &  ci -> ro-
{~lix        -> bx- }       { s   & bx -> cox+,lox+ }       ri & ~ci -> s+
                            {~s        -> cox-,lox- }      ~ri        -> s-
```

The gates are shown in Figure 2. I eliminated the staticizers in the two aC-elements to save space. Unfortunately, this simplification resulted in a race condition, which is described in Section 4.

Turning our attention to communication with the row encoder, A, it is possible to merge this communication with the column communication, $C!b$, as they occur simultaneously. Hence, we can drive `ao` as well as `cox` with `s`, and combine `ci` with `ai` using a C-element. Alternatively, since we activate the row and the encoder at the same time with `s`, we can treat the encoder's outputs just like we treat the neuron's outputs, add a few more bits to the latch to store the row address, and use the column bus's acknowledge `ci` to indicate when both the row's state and address are latched. Merging the column and row-encoder communications (C and A) in this way eliminates the need for a C-element. However, it requires us to compensate for the worst-case delay between the encoding processes and the column communications.

Now let us perform HSE on $\mathsf{BUS}(Y, X)$, making L_x passive and C active.

```
# ({{liyx,},},ci)bus({loy,},{cox,}) {y = 1..Y},{x = 1..X} #
*[{{liyx |} -> cox+,}; [ci]; {{liyx |} -> loy+,};
  {{~liyx &} -> cox-,}; [~ci]; {loy-,};]
```

The implementation is straightforward, as row activity is mutually exclusive. We obtain outgoing data by ORing together all the incoming data for that column. The acknowledge must be steered back to the right row. We determine which row was active by ORing together its data inputs. But wait a minute, why go through all this trouble just to prevent the acknowledge from showing up on the other rows? Actually, we want them to snoop on the bus and wait until ongoing communications terminate before they board. So it is okay to broadcast the acknowledge! Therefore, we can eliminate the Y X-input OR gates that determine input activity and the Y aC-elements that steer the acknowledge to the right row. The only gates needed are a Y-input OR gate for each column and a single wire for the acknowledge.

3.2: Choosing

Arbiters with any number of inputs, N, may be built from two-input arbiter cells using the following recursive definition [3],

$$\mathsf{ARB}(N) \equiv \mathsf{ARB}(N/2) \| \mathsf{ARB}(2) \| \mathsf{ARB}(N - N/2),$$

as shown in Figure 1. A total of $(N - 1)$ $ARB(2)$ cells are required to arbitrate between N inputs; these cells are connected in a balanced binary tree with $\lceil \log_2(N) \rceil$ levels. The two-input arbiter cell is described by:

$$\mathsf{ARB}(2) \equiv *[[\overline{L_1} \rightarrow R; L_1; L_1; R \mid \overline{L_2} \rightarrow R; L_2; L_2; R]]$$

This process probes its L ports to determine if there are active neurons in either its $N/2$-subgroup or its $(N - N/2)$-subgroup. Next, it communicates on its R port to ensure that the group of neurons it serves has been chosen. And finally, it communicates on either of its L ports to select an active subgroup, and ignores the other subgroup. Thus, activity is relayed up the tree by probing the R-to-L channels, while selection is relayed down the tree by communicating on the same channels. Mutual exclusion is guaranteed by performing a second pair of L and R communications to terminate the selection. The two-input arbiter at the top of the tree, which serves all the neurons, is special, since its group is always chosen. Therefore, its R port is connected to a process that automatically completes the communication (i.e., $*[L]$). I present a brief derivation and description of my arbiter design here, which improves on the robustness and performance of prior designs [12, 7, 5, 2]. A complete description is available in [3].

We decompose the two-input arbiter into two subprocesses that handle communications between the two input ports and the common output port, and a third subprocess that

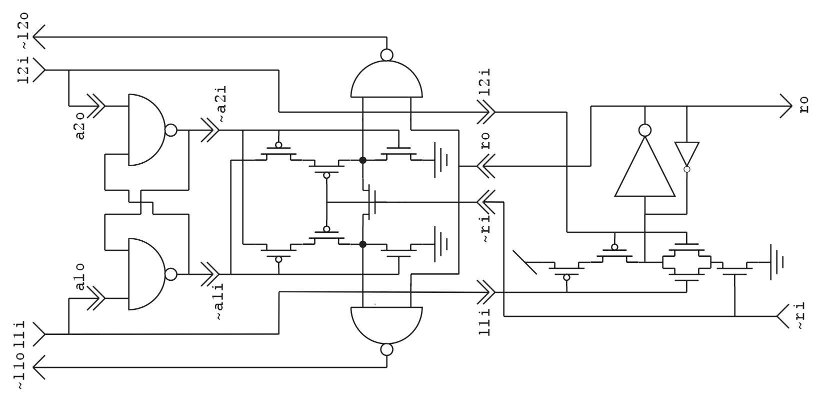


Figure 3: Two-Input Arbiter Circuit

Active-high request signals, `l1i` and `l2i`, propagate down the arbiter tree through a modified OR gate (right) and active-low acknowledge signals, `r̃i`, propagate up the tree through a router (middle). A flip-flop (left) arbitrates between the requests and controls the router. The router filters the flip-flop's metastable (i.e., in-phase) oscillations to ensure the decision is irreversible before it steers the incoming active-low acknowledge to the chosen request, by NORing it with the flip-flops's active-low outputs. Filtering is realized by source-switched PMOS transistors, which cut off power to the pull-ups when the flip-flop's outputs differ by less than the threshold voltage [8]. Since one of the flip-flop's outputs is always high, we connected a single transistor between the outputs, instead of providing a `r̃i`-driven pull-down for each output [12]. A pair of NAND gates invert active-high acknowledges from the steering circuit and blocks them from propagating up the tree when the outgoing active-high request, `ro`, is low.

arbitrates between them. By disentangling arbitration and communication, we obtain the following simpler subprocesses:

$$\begin{aligned} \mathsf{ARB}(2) \equiv {} & *[[\overline{A_1} \rightarrow A_1; A_1 \mid \overline{A_2} \rightarrow A_2; A_2]] \\ & \| *[[\overline{L_1} \rightarrow A_1; R; L_1; L_1; R; A_1]] \| *[[\overline{L_2} \rightarrow A_2; R; L_2; L_2; R; A_2]] \end{aligned}$$

I have specified connectivity implicitly, using the same name for ports that are connected.

We use the reshuffling below for the input/output subprocesses (L_1 and L_2 are passive; A_1, A_2, and R are active); the arbiter itself is implemented in the standard fashion using two cross-coupled NAND gates [8].

```
*[l1i -> a1o+,l1i & ~ri -> ro+; [ri & a1i]; l1o+; [~l1i]; ro-,a1o-; [~a1i]; l1o-;]
```

When a request is received from the lower level, we make a request to the local arbiter and relay the incoming request to the upper level. We do not wait for a response from the arbiter, but we make sure the upper level has cleared its acknowledge to the previous request. If not, we do not make a new request. Instead, we accept the old acknowledge, assuming it is stable (i.e., `ro` is high), and relay it to the lower level as if it was a new acknowledge. However, we wait for a response from the arbiter before we relay this acknowledge to the lower level. At this point, we are half way through the communication cycle, and all signals are set high. When the lower level clears its request, we clear our request to the arbiter. To prevent a new incoming request from using an unstable acknowledge from the arbiter, we wait for the arbiter to clear its acknowledge before we clear our acknowledge to the lower level. Also, when we clear our request to the arbiter, we clear our request to the upper level as well only if both incoming requests have been cleared. A strategy that gives our sister process the opportunity to service her daughters with the old acknowledge. The circuit is shown in Figure 3.

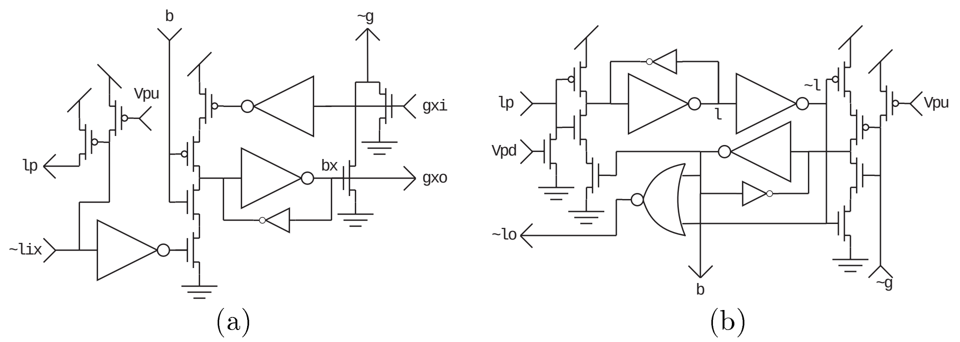


Figure 4: Latch Cell and Control Circuits

(a) This circuit latches data on the column line, `l̃ix`, which is pulled low when the neuron in the selected row is spiking. It sends a request to the interface circuit on `gxo`, which acknowledges on `gxi`. (b) This circuit strobes the row-data/address latch using `b`; monitors write and read–reset operations on the latch using `g`; monitors data transmission on the column lines using `lp`; and tells the row interface circuits when column data is written and the column lines are clear using `l̃o`.

3.3: Latching on

We now direct our attention to latching data from the column bus, and transmitting the stored spikes sequentially by communicating with the column arbiter and the column encoder. We start by isolating communication with the arbiter and the encoder, decomposing the latch's CHP program as follows:

$$\begin{aligned}\mathsf{LATCH}(X) \quad = \quad & *[L?b; \langle \| x : 1..X : [b.x \rightarrow G_x; b.x{\downarrow} \, [\!] \neg b.x \rightarrow \mathsf{skip}]\rangle] \\ & \| * [\langle \| x : 1..X : [\overline{G_x} \rightarrow C_x; A_x \| G_x; C_x)]\rangle]\end{aligned}$$

The second process synchronizes the first process—which is the latch proper—with the arbiter and the encoder; they communicate over their G_x ports. Starting with the first process, making L passive and G_x active yields the following HSE.

```
# ({lix,},{gxi,})latch({gxo,},lo) {x = 1..X} #
*[{lix -> bx+ ,}; {([bx]; gxo+; [gxi]; bx-; [~bx]; gxo-) ||}
||([{bx |}]; lo+; [{~lix &}]; lo-); [{~gxi &} & ~lo];]
```

I acknowledge the column bus (i.e., `lo+`) as soon as any of the bits are set and complete L concurrently with G_x. I postponed the second half of G_x until after $b.x$ is reset, a benign change.

Our immediate goal is to divide the latch proper into three subsequences: `set bits`, `clear bits`, and `sense column lines`—and a fourth subsequence, `control`, that coordinates them. Let us introduce a strobe signal, `b`, a full signal `g`, and a column signal `l` for this purpose. `b` is set when it is safe to write new data into the latch; `g` is set by ORing the stored bits; and `l` is set by ORing the column data signals. Augmenting the sequence above with these variables, we obtain:

```
# ({lix,},{gxi,})latch({gxo,},lo)  {x = 1..X} #
*[({b & lix -> bx+ ,}; {bx |} -> g+)||(b & {lix |} -> l+); [g & l]; b-;
   ({([~b & bx]; gxo+; [gxi]; bx-; [~bx]; gxo-) ||}; {~gxi &} -> g-)
 ||([~b]; lo+; [{~lix &}]; l-); ~g & ~l -> b+, ~l -> lo-;]
```

First, we set the bits and sense the column data concurrently, and confirm completion. Next, we clear the bits and acknowledge the column bus, and confirm completion in this case as well.

It is safe to make the latch transparent again when it is empty and the column data lines have been cleared. However, waiting for the arbiter to clear its last acknowledge first (i.e., `[{~gxi &}]`) allows us to send it new requests as soon as data is written into the latch. Therefore, I wait for `{~gxi &}` to become true before I clear `g`. Whereas, I clear `l` once the column data is cleared (i.e., `[{~lix &}]`). And then I set `b` when both `g` and `l` are cleared—and clear it when they are both set. Whereas, I clear `lo` immediately after `l` is cleared.

We create the opportunity to write data onto the column bus in advance by clearing our acknowledge (i.e., `lo-`) before the latch is emptied (i.e., `[~g]`). Therefore, I clear `lo` and set `b` concurrently. However, deadlock could occur if new data appears—after `lo` goes low—and takes `l` back up while we are waiting for `g` to go low. Including `b` in the guard for `l+` prevents this. On the other hand, I do not set `lo` and clear `b` concurrently. I break the symmetry because we must be sure the write was successful before we clear `lo`, and hence it cannot occur before `g` becomes true. Either we clear `lo` at the same time as `b` or immediately afterwards. I chose the sequential option to avoid introducing an isochronic fork; the speed penalty is minimal.

Hence, we obtain the following four subsequences:

```
# set bits {x = 1..X} #                    # clear bits {x = 1..X} #
*[{([b & lix]; bx+) ||}; [{bx |}]; g+;]    *[{([bx]; gxo+; [~b & gxi]; bx-;
# control #                                    [~bx]; gxo-) ||}; [{~gxi &}]; g-;]
  *[[g & l]; b-; [~g & ~l]; b+;]           # sense column lines #
||*[[~b & l]; lo+; [~l]; lo-;]             *[[b & {lix |}]; l+; [{~lix &}]; l-;]
```

I included `l` in the wait before `lo+` in `control` because it is impossible to tell which half of the sequence we are in without checking `l`; it also eliminates interference between the production rules. I also postponed the `[~b]` wait in `clear bits` till just before `bx-`, giving the request to the arbiter a head start. It is safe because we do not clear the bit when the arbiter acknowledges—we wait till `g` is set and `b` is cleared. As you will see shortly, this reshuffling reduces the `gxo` gate to a wire and eliminates interference in the `bx` gate.

After strengthening and weakening a few guards, these sequences are realized by the following PRS:

```
{ b & lix            -> bx+ }      { bx       -> gxo+ }            l  & ~b  -> lo+
{~b & gxi            -> bx- }      {~bx       -> gxo- }           ~l (|  b) -> lo-
{ bx  |} (| {gxi |}) -> g+           ~g  & ~l  -> b+         { lix |} &  b  -> l+
{~gxi &} (& {~bx &}) -> g-            g  &  l  -> b-         {~lix &}       -> l-
```

I strengthened the guard of `g-` to eliminate interference with `g+`, and I weakened the guard of `g+` to obtain a OR gate. I also weakened the guard of `lo-` to make it an OR gate as well. The gates are shown in Figure 4; `{lix |}` is labeled `lp`.

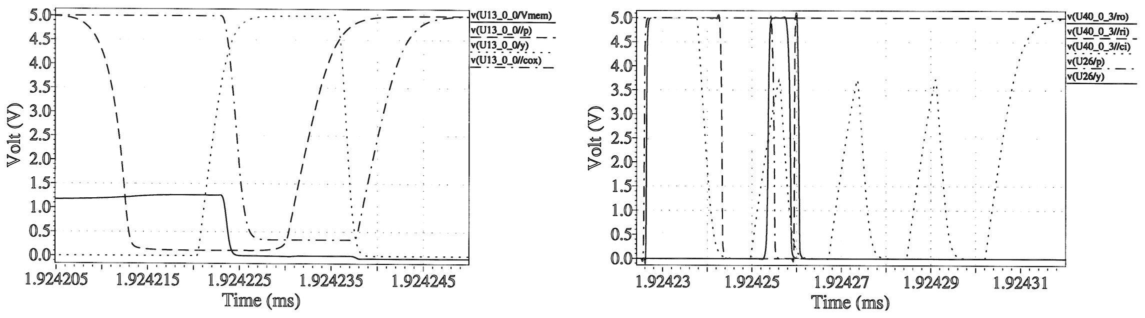
Finally, let us expand the subprocess that communicates with the column arbiter and the column encoder by exploiting its similarity to the row process. We can reuse the HSE sequence we designed to coordinate the row with the row arbiter and the row encoder. Making the substitutions $G_x \Rightarrow L_x$, $C_x \Rightarrow R$, and $A_x \Rightarrow C$, we obtain

```
*[[gxo]; co+; [ci & ~ai]; gxi+,ao+; [~gxo & ai]; co-; [~ci]; gxi-,ao-;]
```

The acknowledge to the latch also serves as the request to the address encoder, as we discussed in Section 3.1. The circuit is thus the same as the one in Figure 2b.

4: Simulation Results

We performed TSPICE (*Tanner Research, Inc.*) simulations to verify the behavior of the design; the netlist was extracted from the layout of a 4×4-pixel imager. However, model parameters for an 0.8μm process were used, whereas the chip was fabricated in a 1.2μm process. We made the interface run freely by tying the request to the acknowledge and feeding current to the neurons.



(a) (b)

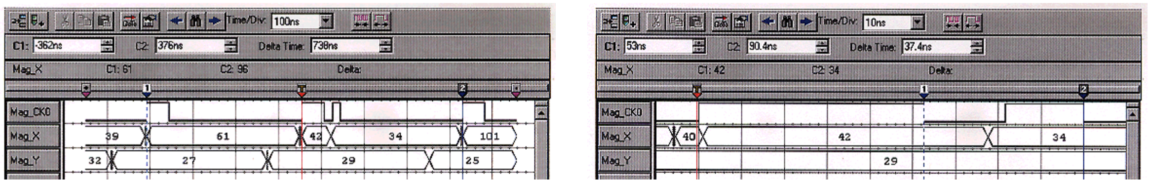
Figure 5: TSPICE Simulation Results

(a) Signals for interface circuit between neuron's (its input voltage is `Vmem`) row request (`p`) and select (`y`) lines, and its column data line (`/cox`). Each major division on the time axis is 5ns, and each minor division is 1ns. (b) Signals for interface circuit between a latch cell's request and acknowledge lines (`p` and `y`), the column arbiter's request and acknowledge lines (`ro` and `/ri`), and the acknowledge from the address encoder (`/ci`), which passes along the receiver's acknowledge. Each major division on the time axis is 10ns, and each minor division is 2ns.

Figure 5a shows the simulated waveforms at the interface between the neuron, the latch cell, and the row's interface circuit—which sends a request to the row arbiter and returns its acknowledge. The communication cycle begins when the neuron's input voltage, `Vmem` (labeled `Iin` in Figure 2a), charged up by analog current from the photodetector, exceeds the threshold, which is about 1.2V. The neuron then makes a request (i.e., `/p` goes low), and, 10ns later, its row is selected (i.e., `y` goes high). The neuron takes 3ns to drive data onto the column line (i.e., `/cox` goes low) and 10ns to withdraw its request. The row-select signal is withdrawn 4ns later, and the neuron takes 3ns to clear its column signal after that.

The rate-limiting steps are selecting a row and clearing the neuron's request; both take 10ns. Selecting a row is slow because it involves arbitration; 6.5ns is spent waiting for the arbiter, to respond. Clearing the row request is slow because of the current-starved nMOS-style inverter. Indeed, the latch controller acknowledges data transfer on the column lines a full 3ns before the row clears its request. However, these slow processes do not impact the cycle-time of the address-event bus because several spikes may be read out in parallel. Thus, the average cycle-time is determined by the column-data transfer cycle-time divided by the number of spikes that were read from the row. And part of the 10ns delay between the neuron's request and row selection may overlap with the previous cycle, since requests may propagate up the arbiter tree while it is selecting another row.

Figure 5b shows simulated waveforms at the interface between the latch cell, the column arbiter, and the column address-encoder—which sends a request to the receiver on the address-event bus and returns its acknowledge. The first two transitions are the request received from the latch cell (labeled `p`) and the request sent to the arbiter (labeled `ro`). The arbiter acknowledges 17ns later (downward transition on `/ri`), and the interface circuit passes the acknowledge on 10ns later (upward transition on `y`). The wait is extended because the arbiter picks another cell first, as you can see from the encoder's active-low acknowledge signal (labeled `/ci`). And the interface circuit waits until `/ci` goes back up before it selects our cell and activates the encoder. When the latch cell is selected, it withdraws its request 1.5ns later. But the interface circuit waits for the encoder to clear its acknowledge before clearing its request to the arbiter 4ns later. The arbiter responds by clearing its acknowledge 1ns later, causing the interface circuit to do the same 1ns later.


Time/Div: 100ns
C1: -362ns C2: 376ns Delta Time: 738ns
Mag_X C1: 61 C2: 96 Delta:

Time/Div: 10ns
C1: 53ns C2: 90.4ns Delta Time: 37.4ns
Mag_X C1: 42 C2: 34 Delta:

(a) (b)

Figure 6: Timing Measurements from 104×96**-neuron Transmitter**

These data were measured with a Tektronix TLA704 Logic Analyzer. In both panels, the top trace is the request, the middle trace is the column address, and the bottom trace is the row address. (a) The first and second address-events shown in this screen-shot are from different rows, whereas the second and third events are from the same row. The cycle time is 362ns in the first case, but it is only 72ns in the second case. (b) Expanded time-scale showing the timing between the second and third event.

These simulation results predict a minimum cycle-time of 18ns (transmitter only); the first event in the burst takes 28ns.

5: Chip testing

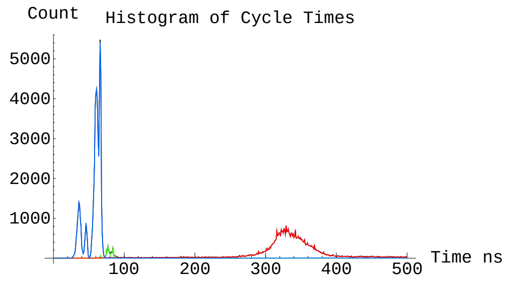
Signals measured from a 104×96-pixel retinomorphic imager with the throughput-on-demand address-event transmitter interface are shown in Figure 6. The chip was fabricated in a 1.2μm CMOS process; it is 9.4mm by 9.7mm. The cycle time is 300-400ns when the event is from a different row that the preceeding one, and it is 30-70ns when it is from the same row. Thus, by reading out an entire row concurrently with sequential transmission of events from the previous row, we were reduced the cycle time by an order of magnitude.

Surprisingly, analysis of the cycle-times between 30ns and 70ns (see Figure 7) revealed no correlation with the number of levels spanned in the arbiter tree:

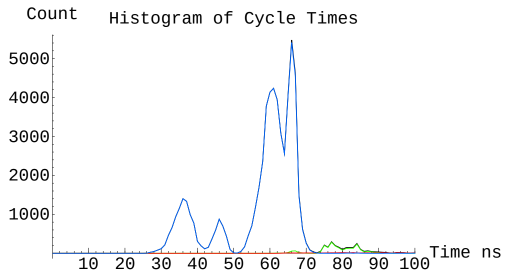
- When all 6 levels of the arbiter tree were spanned, the distribution was strongly (but not purely) unimodal; it peaked at 66ns.
- When only the first level of the arbiter tree was spanned, the distribution was strongly (but not purely) unimodal; it peaked at 36ns.
- And when an intermediate number of levels where spanned, the distribution was multimodal, with peaks at these two times as well as intermediate ones—namely 36ns, 46ns, 61ns and 66ns.

This anomalous behavior may arise because the second event in a burst always takes longer than latter events, irrespective of how many levels are spanned.

In particular, first-to-second-event cycles were unimodally distributed with a peak at 66ns—clustered extremely tightly between 64ns and 72ns. Since this delay is independent of the number of levels of arbitration involved, we surmised that it arose from the latch. In particular, we must wait for the controller to make the latch opaque (by taking `b` low) before we clear any of the bits. This delay extends the first-to-second-event cycles. We also observed that the cycle-time for subsequent events in a burst were sometimes as long as the first-to-second event cycle-time. In particular, the distributions of cycle times for the third-to-fourth and fourth-to-fifth events in a burst had peaks that fell in the 64-72ns range. Another curious observation we made was that adjacent events with the same row and column addresses occur much more frequently than chance allows—giving rise to a peak in the cycle-time distribution around 80ns.



(a)



(b)

Figure 7: Histograms of Cycle Times

For a train of 131,071 events recorded over 93.93ms—a 1.39MHz mean rate—and time-stamped with 0.5ns resolution (bin size is 1ns). (a) There are three distinct clusters: (i) Cycles demarcated by a pair of events with different row and different column addresses cluster around 330ns. (ii) Cycles demarcated by a pair of events with the same row and the same column addresses cluster around 80ns. (iii) Cycles demarcated by a pair of events with the same row address but different column addresses are clustered below 70ns. (b) The time-scale is expanded to resolve multiple peaks in the third cluster, which occur at 66.5ns, 60.6ns, 46.0ns, and 36.0ns.

We suspect the latch controller reads data from several rows while `b` is low, because, seeking simplicity, we mistakenly used a simple NOR gate to sense the column lines instead of the aC-element shown in Figure 4b. The sequence:

```
*[[{lix |}]; l+; [~b & l]; lo+; [{~lix &}]; l-; [~l]; lo-;],
```

obtained by concatenating the `sense column lines` subsequence and the second `control` subsequence (see Section 3.3), and deleting the `b` term, confirms this. As the latch is opaque, all the information read from the rows would be lost. The only evidence would be an extended cycle time due to waiting for `l` to go low before we take `b` high when the latch becomes empty. A plausible explanation why we observe longer cycle times around the fourth event, since (4×70)ns is close to 300ns lower-limit of the row-read cycle.

We also suspect the latch controller sends the same spike several times while `l` is low, because, seeking simplicity, we mistakenly NORed `l` and `g` to obtain `b` instead of using a C-element shown in Figure 4b. The sequence:

```
*[[b & lix]; bx+; [bx]; g+; [g | l]; b-;..; [~b & gxi]; bx-;..; g-; [~g & ~l]; b+;]
```

obtained by concatenating the `set bits` subsequence, the `clear bits` subsequence, and the first `control` subsequence (see Section 3.3), and replacing `[g & l]` with `[g | l]` confirms this. If the time it takes for `l` to go high when data is placed on the column lines is longer than the time it takes to read and send one spike—but less than the time to send two—the data lines would be read again when only one data bit is set. A plausible explanation for the frequent occurrence of adjacent events with the same row and column addresses separated by 80ns.

And we discovered a race condition in the neuron's interface circuit, caused by eliminating the staticizers in the pixel's aC-elements to save space.[3] It occurs when the row is selected and the first aC-element is disabled (see Figure 2). At this point, if `~lix` has not discharged all the way to ground, the pull-down continues to pass current and resets `bx`. Failure occurs if `bx` is reset before the output of the gate it drives has transitioned. And even if it has, we have an unstable request (i.e., `~p`) that can disappear at any time. We can fix this problem by introducing another inverter stage to produce a faster transition. We can avoid adding transistors by making the fourth gate in the chain an aC-element and

[3]Charles Higgins discovered this race condition and brought it to my attention.

the second one a simple inverter.

6: Conclusions

I have described a scalable 2-D address-event transmitter interface that was designed to take advantage of the high integration densities available with advanced submicron technology. To sustain throughput, it exploits the linear increase in the number of active neurons per row with increasing array size by reading spikes in parallel. I synthesized an asynchronous implementation starting from a high level specification, and I presented test results from a 104×96-neuron interface fabricated in a 1.2μm process. Reading the state of all neurons in a selected row in parallel reduced the cycle time from 300-400ns to 40-70ns—six to ten times shorter than the 420ns minimum cycle time reported previously [2].

7: Acknowledgments

This work begun while I was a doctoral student at the California Institute of Technology, in Carver Mead's lab, where it was funded by the Office of Naval Research; DARPA; and the National Science Foundation Engineering Research Center Program. It is currently funded by start-up funds provided by the Schools of Engineering and Medicine (through the Institute of Medicine and Engineering) at the University of Pennsylvania. I thank Chuck Higgins and Tim Horiuchi for their invaluable help beta-testing this interface, and fishing out and documenting bugs.

References

[1] K A Boahen. The retinomorphic approach: Pixel-parallel adaptive amplification, filtering, and quantization. *Analog Integr. Circ. and Sig. Proc.*, 13:53–68, 1997.

[2] K A Boahen. *Communicating Neuronal Ensembles between Neuromorphic Chips*, chapter 11, pages 229–262. Neuromorphic Systems Engineering: Neural Networks in Silicon. Kluwer Academic Pub., Boston MA, 1998.

[3] K A Boahen. Massive connectivity between neuromorphic chips using address-events. *To appear in IEEE Trans. Circ. & Sys.*, VVV:xxx–yyy, 1999.

[4] C M Higgins and C Koch. Multi-chip motion processing. In *Conference on Advanced Research in VLSI*, Los Alamitos, CA, 1999. IEEE Computer Society Press.

[5] 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.

[6] J P Lazzaro. Low-power silicon spiking neurons and axons. In *IEEE International Symposium on Circuits and Systems*. IEEE Press, 1992.

[7] M Mahowald. *An Analog VLSI Stereoscopic Vision System*. Kluwer Academic Pub., Boston, MA, 1994.

[8] A Martin. Programming in vlsi: From communicating processes to delay-insensitive circuits. Technical Report CS-TR-89-01, California Institute of Technology, Pasadena CA, 1989.

[9] C A Mead. *Analog VLSI and Neural Systems*. Addison Wesley, Reading MA, 1989.

[10] 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.

[11] A Murray and L Tarassenko. *Analogue Neural VLSI: A Pulse Stream Approach*. Chapman and Hall, London, England, 1994.

[12] M Sivilotti. *Wiring considerations in Analog VLSI Systems, with application to Field-Programmable Networks*. PhD thesis, California Institute of Technology, Pasadena CA, 1991.