# A Serial H-Tree Router for Two-Dimensional Arrays

Sam Fok and Kwabena Boahen
Stanford University
Stanford, CA 94305, USA
{samfok,boahen}@stanford.edu

***Abstract*— Existing routing mechanisms for two-dimensional (2D) arrays either use low-overhead grids with one or two shared wires per row or column (e.g., RAM) or high-overhead meshes with many wires connecting neighboring clients (e.g., supercomputers). Neither is suitable for intermediate-complexity clients (e.g., small clusters of silicon neurons). We present a router tailored to 2D arrays of such clients. It uses a tree laid out in a fractal pattern (H-tree), which requires less wiring per signal than a grid, and adopts serial-signaling, which keeps link-width constant, regardless of payload size. To route from the tree's leaves to its root (or vise versa), each node prepends (consumes) a delay-insensitive 1-of-4 code that signals the route's previous (next) branch; additional codes carry payload. We employ this serial H-tree router to service a 16×16 array of silicon-neuron clusters, each with 16 spike-generating analog somas, 4 spike-consuming analog synapses, and one 128-bit SRAM. Fabricated in a 28-nm CMOS process, the router communicates 26.8M soma-generated and 18.3M synapse-targeted spikes per second while occupying 43% of the client's 35.1×36.1$\mu$m$^2$.**

## I. Router Functionality and Overhead

Advances in CMOS fabrication processes enable increases in the number and complexity of computational units in highly distributed and parallel architectures (e.g., neuromorphic processors; [1]–[3]), which calls for a corresponding increase in scalability and sophistication of routing mechanisms. Router area should be a reasonable fraction of the total system—router architecture is therefore dictated by client complexity. In this regard, high-overhead routers (e.g., meshes with parallel interfaces) capable of communicating arbitrary data-types at high bandwidths, are unsuited for intermediate-complexity clients with lower data-rate requirements. To provide multiple-data-type functionality for such clients, we adapt existing low-overhead routers.

Low-overhead routers contain a transmitter and a receiver [4]. The transmitter merges data from all of the clients into a single stream and adds source-identifying addresses to each datum to form a packet. The receiver takes a stream of packets, parses each destination-identifying address, and delivers the datum to the specified client.

For $N$ clients, a low-overhead router's circuitry scales as $\mathcal{O}(\sqrt{N})$ by sharing resources. Clients are tiled in a two-dimensional (2D) array and share row and column wires within the array and transceiver circuitry at the edge of the array [4], [5] (Fig. 1, GRID ADDR). Sharing works correctly if certain timing assumptions are met [6], [7], but these assumptions are difficult to satisfy for long wires, which are susceptible to phenomena such as *charge relaxation*, whereby a significant voltage difference arises between the wire's two ends [8]. For this reason, grids do not readily scale to large arrays.

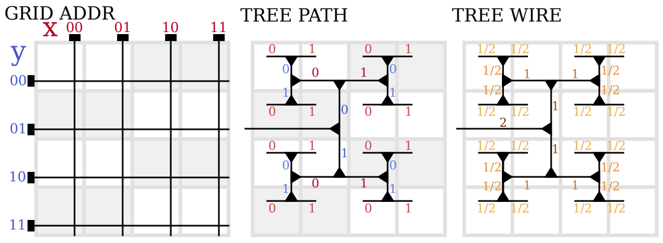

Fig. 1. Grid Addresses and Tree Paths: clients (white and gray squares) are tiled in a 2D array and routed to (or from) using a grid or a tree. GRID ADDR: A client's address is encoded by concatenating its $x$ and $y$ positions (in binary). Addressing circuitry is placed at the array's edge (black rectangles) and scales as $\mathcal{O}(\sqrt{N})$ for $N$ clients ($N = 16$ shown). TREE PATH: A client's path is encoded by traversing the tree from the root to leaf (indexed by $n = 3$ and $n = 0$, respectively, in Algorithms 1 & 2). Each up and left (down and right) branch appends a 0 (1). Shaded squares indicate differences between tree and grid binary-number assignments (e.g., the bottom left client's grid-address is 0011, but its tree-path is 1010). Routing circuitry is embedded within the array (black triangles) and scales as $\mathcal{O}(N)$. TREE WIRE: Wire segments are annotated with their lengths.

Our router presented herein switches from grid addresses to tree paths (Fig. 1, TREE PATH), trading an increase in logic circuitry for enhanced scalabilty and functionality. The increase in logic circuitry—from $\mathcal{O}(\sqrt{N})$ to $\mathcal{O}(N)$ for $N$ clients—is worthwhile for emerging intermediate-complexity clients that use thick-oxide transistors for ultra-low power analog computation and much smaller thin-oxide transistors for ultra-fast digital communication [9]. The enhanced scalability arises because its asynchronous implementation's timing assumptions are easily met. And the enhanced functionality arises because its serial protocol supports multiple datatypes, whereas the grid's parallel protocol limits payload size. For backward compatibility, converting grid addresses into tree paths and vice versa is straightforward (Algorithms 1 & 2).

In Section II, we describe how grid addresses and tree paths are encoded, show that both require $\mathcal{O}(N)$ wiring, and justify our choice of a 4-ary tree over a binary tree. In Section III, we describe the serial link our router uses. In Section IV, we describe the logical design of the router's nodes. In Section V, we describe how the router's leaves were customized for a neuromorphic application. In Section VI, we describe the router's logical and physical synthesis and its verification and validation. In Section VII, we conclude the paper with a discussion of our results.

**Algorithm 1** Converts Path $(p)$ to Address $(x,y)$

```
Require: $l = \text{length}(p)$
  for $n = 0$ to $l/2 - 1$ do
    $x[n] \leftarrow p[2n]$
    $y[n] \leftarrow p[2n+1]$
  end for
```

**Algorithm 2** Converts Address $(x,y)$ to Path $(p)$

```
Require: $l = \text{length}(p)$
  for $n = 0$ to $l/2 - 1$ do
    $p[2n] \leftarrow x[n]$
    $p[2n+1] \leftarrow y[n]$
  end for
```

## II. Tree Paths versus Grid Addresses

A 2D array can be routed to (or from) using a grid or a tree. We consider $N$ clients, of unit width and height, arranged on a square grid (Fig. 1, WIRING).

Given equal link-widths, the tree requires less wiring than the grid. To calculate the length of the tree's wiring, $W_t$, we start from the $N$ unit centers: $N$ $\frac{1}{2}$-unit segments project horizontally from each center. At the second level, $\frac{N}{2}$ $\frac{1}{2}$-unit segments project vertically. At the third level, $\frac{N}{4}$ 1-unit segments project horizontally. This geometric pattern continues up to the root; each level alternates between horizontal and vertical orientation and halves the number of segments from the previous, lower level, while doubling the segment-length every other level. Overall, we have

$$W_{\mathrm{t}} = \frac{1}{2}\left(N + \frac{N}{2}\right) + 1\left(\frac{N}{4} + \frac{N}{8}\right) + 2\left(\frac{N}{16} + \frac{N}{32}\right) + \ldots = \frac{3}{2}N$$

as $N$ scales up (coloring matches Fig. 1, WIRING). For comparison, in the grid, each client adds 2 units of wire so that $W_{\mathrm{g}} = 2N$. Therefore, $W_{\mathrm{t}} = \frac{3}{4}W_{\mathrm{g}}$: the tree uses up to 25% less wiring than the grid.[1] While the tree's segments become longer as we move from leaves to root, they are shared among more and more leaves.

However, the primary trade-off is the tree's larger transistor-count ($\mathcal{O}(N)$ versus $\mathcal{O}(\sqrt{N})$ for the grid), determined by the node-count times the transistors-per-node. To reduce the node-count, we opted for a 4-ary tree over a binary tree. A binary tree has $N-1$ nodes whereas a 4-ary tree has $\frac{N-1}{3}$ nodes. In general, a $k$-ary tree has $\frac{N-1}{k-1}$ nodes and $\log_k(N)$ levels. Consequently, switching divides the node-count by three, halves the number of levels, halves the latency, and doubles the unpipelined throughput.

If switching from binary to 4-ary doubles the transistors-per-node,[2] and divides the node count by three, we would expect to decrease the overall transistor count by 33%. However, nodes are not homogeneous; leaf nodes are tailored to clients' needs. The total transistor count in an $N$-client $k$-ary tree whose leaf and intermediate nodes have $T_{\mathrm{L}k}$ and $T_{\mathrm{I}k}$ transistors each, respectively, is

$$T_{\mathrm{tot}k} = \frac{N-1}{k-1}\,\frac{(1-k/N)T_{\mathrm{I}k} + (k-1)T_{\mathrm{L}k}}{k - k/N} \qquad (1)$$

[1]In emerging 3D processes, with wire-segments traveling along three axes, segment-count still halves at each level towards the root, but segment-length only doubles every third level (c.f. every other level in 2D). As a result, $W_{\mathrm{t}} = \frac{7}{24}W_{\mathrm{g}} \approx 0.29W_{\mathrm{g}}$: the tree uses up to 71% less wiring.

[2]Doubling occurs if combinational gates (e.g., NANDs or NORs)—whose transistor count is 2× their fan-in—dominate. For sequential gates—whose state-holding elements are not replicated—the increase is sublinear. When gates are treed to build wider gates, the increase is supralinear.

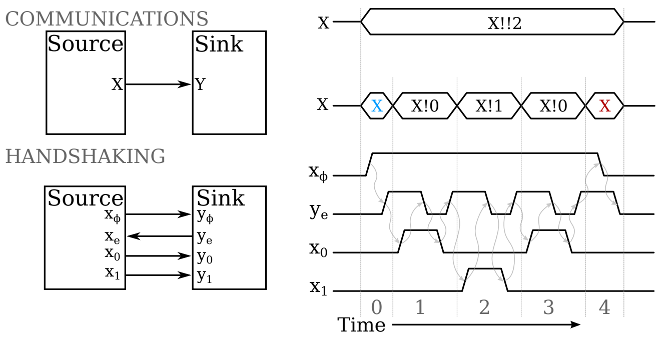

Fig. 2. Serial Link Description at Two Levels of Abstraction. HANDSHAKING: *Source* drives control line $x_\phi$ and data lines $x_0$ and $x_1$. *Sink* drives control line $y_e$. Arrows point from driver to listener. Two-phase handshakes (time slots 0 and 4) initiate and terminate packet communication; four-phase handshakes (1, 2, and 3) send the packet's bits as 1-of-2 codes. All transitions are acknowledged (curved gray arrows), so the protocol is delay-insensitive. $y_e$'s last transition is acknowledged by $x_\phi$'s initial transition in the next packet. A three-bit packet (010) is communicated in this example, but the protocol supports arbitrary-sized packets and 1-of-$D$ codes using $D$ data lines. COMMUNICATIONS: A channel connects *Source*'s output port ($X$) to *Sink*'s input port ($Y$). *Source*'s dataless communications ($X$ and $X$) initiate and terminate packet transmission; such communications are colored blue and red, respectively, throughout the text. Its datafull communications ($X!0$ and $X!1$) send the packet's bits. The entire communication sequence may be consolidated into the single operation ($X!!2$).

| | Transmitter | | Receiver | |
|---|---|---|---|---|
| $k$ | 2 | 4 | 2 | 4 |
| $T_{\mathrm{L}k}$ | 78 | 208 | 30 | 54 |
| $T_{\mathrm{I}k}$ | 91 | 255 | 64 | 148 |
| $T_{\mathrm{tot4}}/T_{\mathrm{tot2}}$ | 0.867 | | 0.550 | |

TABLE I

NODE TRANSISTOR COUNTS & BINARY:4-ARY RATIOS FOR LARGE $N$

For $T_{\mathrm{L}k} = T_{\mathrm{I}k} = T_k$, this expression reduces to $\frac{N-1}{k-1}T_k$: the total number of nodes times the transistors-per-node. Note that the ratio of leaf to intermedaite nodes is $k-1{:}1-k/N$, which approaches 1:1 and 3:1 for binary and 4-ary trees, respectively, as $N$ increases. Thus, based on $T_{\mathrm{L}k}$'s and $T_{\mathrm{I}k}$'s values for our designs (Tab. I),[3] which have different mixes of combinational and sequential logic and treed gates, switching from binary to 4-ary increases the average transistor count of the transmitter's and receiver's nodes by 2.6× and 1.6×, respectively. As a result, their overall transistor count reduces by 13.3% and 45%, respectively (see Tab. I).

We built our serial tree-router with quasi-delay insensitive (QDI) circuits. The only timing assumption made is the *isochronic fork*. Signal-propagation delay along branches of such forks are assumed to be *equally* insignificant (hence the *iso* combining form; precise definitions may be found in [10], [11]). This assumption is the minimal one necessary for useful computation with asynchronous circuits (i.e., Turing complete). No assumptions are made about signal-propagation delays through gates or nonisochronic wires, except that they are positive and finite.

## III. Serial Communication Protocol

To keep link-width constant, we use serial communication. The path-length grows as we move from leaf to root in a tree.

[3]The leaf node's communication is dataless—it requests or acknowledges.

Hence, codes communicated over links closer to the root have more bits than those communicated over links closer to the leaves. A parallel protocol thus requires wider links (i.e. more wires) towards the root, whereas a serial protocol makes do with a constant width. Further, the latter allows us to communicate more than just the encoded paths; we can communicate data (e.g., configuration settings) as well.

Our serial-link follows a fully delay-insensitive version of the *bundled-data* protocol in [7] (Fig. 2, HANDSHAKING). For example, the following source generates a random bit-stream and segments it into packets of arbitrary length (see Table II for syntax):

$$\begin{aligned} x_\phi{\uparrow};\, [x_e];{*}[[\text{true} &\longrightarrow x_0{\uparrow};\, [\neg x_e];\, x_0{\downarrow};\, [x_e] \\ |\,\text{true} &\longrightarrow x_1{\uparrow};\, [\neg x_e];\, x_1{\downarrow};\, [x_e] \\ |\,\text{true} &\longrightarrow x_\phi{\downarrow};\, [\neg x_e];x_\phi{\uparrow};\, [x_e]]] \end{aligned}$$

*Handshakes* that demarcate the beginning and end of packet transmission are colored blue and red, respectively. If the $x_\phi$ branch is executed immediately, or consecutively, the packet contains no data. A sink that consumes the source's data operates as follows.

$$\begin{aligned} [y_\phi];\, y_e{\uparrow};{*}[[y_0 \vee y_1 &\longrightarrow y_e{\downarrow};\, [\neg y_0 \wedge \neg y_1];\, y_e{\uparrow} \\ \square \quad \neg y_\phi &\longrightarrow y_e{\downarrow};[y_\phi];\, y_e{\uparrow}]] \end{aligned}$$

Selection (deterministic) is used instead of arbitration (non-deterministic) because the source guarantees mutual exclusion between the branches.

At a higher level of abstraction, we describe the source's and sink's operation simply in terms of *communications* on ports connected by a channel (Fig. 2, COMMUNICATIONS). For the source:

$$X;{*}[[\text{true} \longrightarrow X!0 \,|\, \text{true} \longrightarrow X!1 \,|\, \text{true} \longrightarrow X;X]]$$

$X$ and $X$ correspond to two-phase handshakes (on $x_\phi$ and $x_e$) that demarcate the packet (see Table III for notation). $X!$ corresponds to repeated four-phase handshakes (on $x_{0,1}$ and $x_e$) that send the payload. For the sink:

$$Y;{*}[[\overline{Y?} \longrightarrow Y? \square \overline{Y} \longrightarrow Y;Y]]$$

Note the unconventional use of the probe to check whether a *datafull* communication is pending. This probe ($\overline{Y?}$) corresponds to $y_0 \vee y_1$, whereas the *dataless* communication probe ($\overline{Y}$) corresponds to $\neg y_\phi$.

We introduce $??$ and $!!$ operators to describe serial read and write communications concisely (Fig. 2, COMMUNICATIONS). The source and sink are described as

$${*}[[\text{true} \longrightarrow X!!\text{null} \,|\, \text{true} \longrightarrow X!!\text{Rand}()]] \parallel {*}[[Y??]]$$

where null is an empty string (i.e. the packet is empty) and Rand() returns a random, nonnegative integer.

## IV. ROUTER LOGICAL DESIGN

The router consists of a transmitter and a receiver, both composed of a tree of nodes (Fig. 3). The transmitter merges packets from the clients into a single stream for transmission to the environment. The receiver does the inverse; it splits each packet in the stream off to the targeted client. For conciseness, we describe the nodes' operation for a binary tree (TX(2) and RV(2)). It is straightforward to extend these processes to a $k$-ary tree (TX($k$) and RV($k$)).

### A. Transmitter

A transmitter node merges packet streams from its children into a single packet stream for its parent (another node one level closer to the root, unless the node is itself the root):

$$\begin{aligned} &\text{TX}(2) \\ &\equiv {*}[[\overline{C_0} \longrightarrow P!!(0 \oplus C_0??) \,|\, \overline{C_1} \longrightarrow P!!(1 \oplus C_1??)]] \end{aligned}$$

A packet from port $C_{0,1}$ is interpreted as a string; $a \oplus b$ prepends $a$ to $b$ (e.g., $1 \oplus 01 = 101$). Prepending a child's port index at that node in the tree to its data builds the overall path from leaf to root.

Expanding $!!$ and $??$ operators, and separating out arbitration, TX(2) becomes

$$\begin{aligned} &{*}[[\overline{C_0} \longrightarrow P;P!0;\, C_0;[\overline{C_0} \longrightarrow P;\, C_0] \\ &\quad |\,\overline{C_1} \longrightarrow P;P!1;\, C_1;[\overline{C_1} \longrightarrow P;\, C_1]] \\ &\parallel {*}[[\overline{C_0?} \longrightarrow P!C_0?\square\overline{C_1?} \longrightarrow P!C_1?]] \end{aligned}$$

| | |
|---|---|
| | **Assignment** |
| $x{\uparrow}$ / $x{\downarrow}$ | Set boolean variable $x$ to true / false |
| | **Program Composition** |
| $s_1;\, s_2$ | Execute segment $s_1$ and then $s_2$ |
| $s_1,\, s_2$ | Execute $s_1$ concurrently with $s_2$ |
| ${*}[s]$ | Execute $s$ repeatedly |
| | **Boolean Operations** |
| $x$ / $\neg x$ | Return the value of $x$ / negated value of $x$ |
| $e_1 \wedge e_2$ | Return the logical-and of $e_1$ and $e_2$ |
| $e_1 \vee e_2$ | Return the logical-or of $e_1$ and $e_2$ |
| | **Branching** |
| $[e]$ | Wait until boolean expression $e$ is true |
| $[e_1 \rightarrow s_1]$ | When $e_1$ becomes true, execute $s_1$ |
| $[e_1 \rightarrow s_1 \,|\, e_2 \rightarrow s_2]$ | If boolean $e_1$ ($e_2$) is true, execute $s_1$ ($s_2$) |
| | If both are true, execute either $s_1$ or $s_2$ |
| | If both are false, wait |
| $[e_1 \rightarrow s_1 \square e_2 \rightarrow s_2]$ | If boolean $e_1$ ($e_2$) is true, execute $s_1$ ($s_2$) |
| | Assume $e_1$ and $e_2$ cannot both be true |
| | If both are false, wait |

TABLE II
HANDSHAKING EXPANSION (HSE) SYNTAX

| | |
|---|---|
| | **Assignment** |
| $x := d$ | Set variable $x$ to $d$'s value |
| | **Communication** |
| $X$ | Communicate on port $X$ (dataless) |
| $X!x$ | Write value of $x$ to $X$ |
| $X?x$ | Read value from $X$ to $x$ |
| $Y!X?$ | Read value from $X$ and write it to port $Y$ |
| $\overline{X}$ | *True* if a communication is pending and *false* if not |
| | **Program Composition** |
| $S_1;\, S_2$ | Execute segment $S_1$ and then $S_2$ |
| $S_1 \parallel S_2$ | Execute $S_1$ in parallel with $S_2$ |
| $S_1 \bullet S_2$ | Overlap the execution of $S_1$ and $S_2$ (called *bullet*) |
| ${*}[S]$ | Execute $S$ repeatedly |
| | **Boolean Operations** |
| $\neg,\ \wedge,\ \vee$ | Same as in Table II |
| $x = d$ | Return *true* if $x$'s value equals $d$'s and *false* if not |
| | **Branching** |
| $\rightarrow,\ |,\ \square$ | Same as in Table II |

TABLE III
COMMUNICATING HARDWARE PROCESSES (CHP) SYNTAX

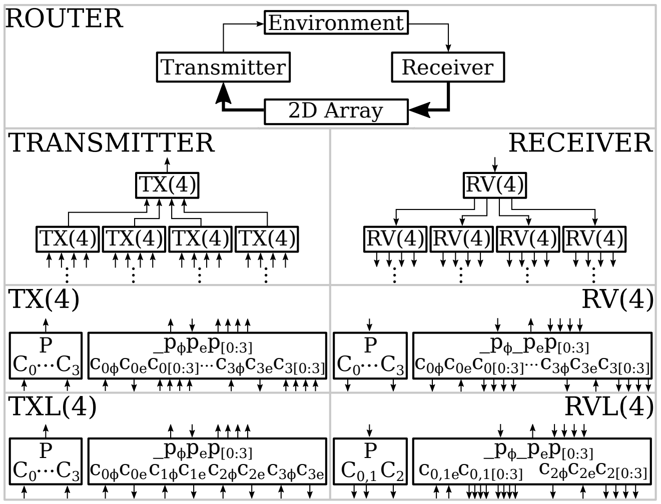

Fig. 3. Router Process Decomposition. ROUTER: Facilitates communication between clients tiled in a 2D array and an external environment using a transmitter and a receiver. TRANSMITTER and RECEIVER: A pair of 4-ary trees provide an input and output port at their leaves for each client. TX(4) and RV(4): CHP ports (left) and HSE signals (right) that interface processes running in TRANSMITTER's and RECEIVER's nodes with their environment. TXL(4) and RVL(4): Same as previous but for processes in the leaves.

Communications that demarcate when packet transmission begins and ends at the child and parent ports are colored blue and red, respectively. Putting $P!\{0,1\}$ before $C_{0,1}$ ensures that the child's index is prepended to the packet before the child's data are forwarded with the $P!C_{0,1}?$ communications.

Further expansion yields the following HSE.

$$
\begin{aligned}
*[[\,& c_{0\phi} \longrightarrow p_\phi\uparrow;\ [p_e];p_0\uparrow;\ [\neg p_e];\ p_0\downarrow;\ [p_e];\ c_{0e}\uparrow;\\
&\quad [\neg c_{0\phi}];\ p_\phi\downarrow;\ [\neg p_e];\ c_{0e}\downarrow\\
\mid\ & c_{1\phi} \longrightarrow p_\phi\uparrow;\ [p_e];p_1\uparrow;\ [\neg p_e];\ p_1\downarrow;\ [p_e];\ c_{1e}\uparrow;\\
&\quad [\neg c_{1\phi}];\ p_\phi\downarrow;\ [\neg p_e];\ c_{1e}\downarrow]],\\
*[[\,& c_{00} \longrightarrow p_0\uparrow;\ [\neg p_e];\ c_{0e}\downarrow;\ [\neg c_{00}];\ p_0\downarrow;\ [p_e];\ c_{0e}\uparrow\\
\square\ & c_{01} \longrightarrow p_1\uparrow;\ [\neg p_e];\ c_{1e}\downarrow;\ [\neg c_{01}];\ p_1\downarrow;\ [p_e];\ c_{1e}\uparrow\\
\square\ & c_{10} \longrightarrow p_0\uparrow;\ [\neg p_e];\ c_{0e}\downarrow;\ [\neg c_{10}];\ p_0\downarrow;\ [p_e];\ c_{0e}\uparrow\\
\square\ & c_{11} \longrightarrow p_1\uparrow;\ [\neg p_e];\ c_{1e}\downarrow;\ [\neg c_{11}];\ p_1\downarrow;\ [p_e];\ c_{1e}\uparrow]]
\end{aligned}
$$

Note that the initial parent communication completes ($[p_e]$) and a code is transmitted to the parent before the initial child communication is acknowledged ($c_{0e}\uparrow$ or $c_{1e}\uparrow$). After that, the selection process relays the child's data.

We proceed by factorizing the arbitration process into the arbiter itself and the remaining child-parent communication:

$$
\begin{aligned}
&*[[c_{0\phi} \longrightarrow s_0\uparrow;\ [\neg c_{0\phi}];\ s_0\downarrow \mid c_{1\phi} \longrightarrow s_1\uparrow;\ [\neg c_{1\phi}];\ s_1\downarrow]],\\
&*[[s_0 \wedge \neg u \longrightarrow p_\phi\uparrow;\ [p_e];w_0\uparrow;\ p_0\uparrow;\ [\neg p_e];\ u\uparrow;\ w_0\downarrow;\\
&\qquad p_0\downarrow;\ [p_e];\ c_{0e}\uparrow;\ [\neg s_0];\ p_\phi\downarrow;\ [\neg p_e];\ c_{0e}\downarrow;u\downarrow\\
&\ \ \square\, s_1 \wedge \neg u \longrightarrow p_\phi\uparrow;\ [p_e];w_1\uparrow;\ p_1\uparrow;\ [\neg p_e];\ u\uparrow;\ w_1\downarrow;\\
&\qquad p_1\downarrow;\ [p_e];\ c_{1e}\uparrow;\ [\neg s_1];\ p_\phi\downarrow;\ [\neg p_e];\ c_{1e}\downarrow;u\downarrow]]
\end{aligned}
$$

$s_{0,1}$ are introduced to store the selection result; $w_{0,1}$ are introduced to distinguish the state immediately after $[p_e]$ (prepending the index) from that immediately after $[p_e]$ (acknowledging the child); and $u$ is introduced to preserve mutual exclusion in the selection process when its branches are implemented as concurrent processes. It prevents the $s_1$ branch (i.e. $p_\phi\uparrow;\ [p_e];\ldots$) from beginning before the $s_0$ branch completes (i.e. $p_\phi\downarrow;\ [\neg p_e];\ldots$) when $c_{1\phi}$ is high and the arbiter executes $s_1\uparrow$ immediately after $s_0\downarrow$.

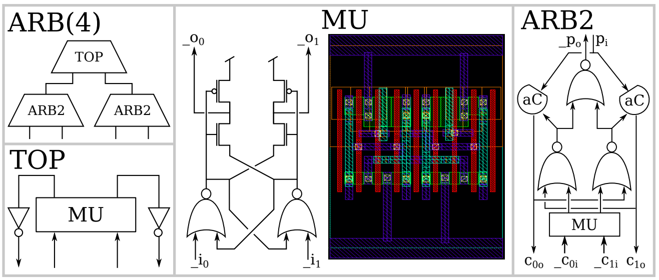

Fig. 4. Four-Way Arbiter. ARB(4): Selects one of four clients with one TOP and two ARB2s; $k$ clients require $k-2$ ARB2s connected in a binary tree. TOP: Performs two-way selection with a MU. MU: Selects one of two active-low (indicated by underscore prefix) inputs ($\_i_0$ and $\_i_1$) using cross-coupled NOR gates. Four additional transistors filter out metastable signals before toggling the outputs ($\_o_0$ and $\_o_1$). MU's custom standard-cell layout is shown. ARB2: Relays its childrens' requests to its parent and relays its parent's grant to a requesting child, selected beforehand by MU. The two, lower NOR gates ensure that handshakes on $c_{0\mathrm{i,o}}$ and $c_{1\mathrm{i,o}}$ do not overlap; aC are asymmetric C-elements.

Our 4-ary transmitter tree's node uses a four-way arbiter (Fig. 4). Three mutual-exclusion elements are interconnected in a binary decision-tree by handshaking circuitry ($c_{0:3\phi}$ and $s_{0:3}$ connect to the two ARB2's $\_c_{0,1\mathrm{i}}$ inputs and $c_{0,1\mathrm{o}}$ outputs, respectively) [12]. CHP and HSE are omitted for brevity. For comparison, a binary tree's node requires just one mutual-exclusion element—with no additional overhead.

The transmitter node's HSE (sans ARB(4)) is implemented by the following production rule set (PRS).

$$
\begin{aligned}
\neg u \wedge (s_0 \vee s_1) &\rightarrow p_\phi\uparrow\\
(c_{0e} \wedge \neg s_0) \vee (c_{1e} \wedge \neg s_1) &\rightarrow p_\phi\downarrow
\end{aligned}
$$

$$
\begin{aligned}
s_0 \wedge p_e \wedge \neg u &\rightarrow w_0\uparrow & u &\rightarrow w_0\downarrow\\
s_1 \wedge p_e \wedge \neg u &\rightarrow w_1\uparrow & u &\rightarrow w_1\downarrow
\end{aligned}
$$

$$
\begin{aligned}
c_{00} \vee c_{10} \vee w_0 &\rightarrow p_0\uparrow & \neg(c_{00} \vee c_{10} \vee w_0) &\rightarrow p_0\downarrow\\
c_{01} \vee c_{11} \vee w_1 &\rightarrow p_1\uparrow & \neg(c_{01} \vee c_{11} \vee w_1) &\rightarrow p_1\downarrow
\end{aligned}
$$

$$
(w_0 \vee w_1) \wedge \neg p_e \rightarrow u\uparrow \qquad \neg(c_{0e} \vee c_{1e} \vee p_\phi) \rightarrow u\downarrow
$$

$$
\begin{aligned}
s_0 \wedge u \wedge p_e \wedge \neg c_{1e} &\rightarrow c_{0e}\uparrow & \neg p_e &\rightarrow c_{0e}\downarrow\\
s_1 \wedge u \wedge p_e \wedge \neg c_{0e} &\rightarrow c_{1e}\uparrow & \neg p_e &\rightarrow c_{1e}\downarrow
\end{aligned}
$$

## B. *Receiver*

A receiver node splits a packet stream from its parent into packet streams for its children (another node one level closer to the leaves, unless the node is a leaf itself):

$$
\begin{aligned}
&\mathrm{RV}(2)\\
&\equiv *[[P??(s,d) \bullet [s=0 \longrightarrow C_0!!d \,\square\, s=1 \longrightarrow C_1!!d]]]
\end{aligned}
$$

It uses the packet's first word (written into $s$) to decide which child to send the remainder of the packet (written into $d$); $s$ has 1 bit for a binary tree or 2 bits for a 4-ary tree (RV(4)).

We expand RV(2)'s ?? and !! communications as follows.

$$\begin{array}{l}
P;P?s \bullet [s=0 \longrightarrow C_0 \,[]\, s=1 \longrightarrow C_1];\\
*[[\overline{P?} \wedge s=0 \longrightarrow C_0!P?\\
\quad []\,\overline{P?} \wedge s=1 \longrightarrow C_1!P?\\
\quad []\,\overline{P} \longrightarrow P\bullet[s=0 \longrightarrow C_0 \,[]\, s=1 \longrightarrow C_1];\\
\qquad P;P?s \bullet [s=0 \longrightarrow C_0 \,[]\, s=1 \longrightarrow C_1]]]
\end{array}$$

This process can be expanded further as

$$\begin{array}{l}
*[[p_0 \wedge s_0 \longrightarrow c_{00}\uparrow;\, [\neg c_{0e}];\, p_e\downarrow;\, [\neg p_0];\, c_{00}\downarrow;\, [c_{0e}];\, p_e\uparrow\\
\quad []\, p_1 \wedge s_0 \longrightarrow c_{01}\uparrow;\, [\neg c_{0e}];\, p_e\downarrow;\, [\neg p_1];\, c_{01}\downarrow;\, [c_{0e}];\, p_e\uparrow\\
\quad []\, p_0 \wedge s_1 \longrightarrow c_{10}\uparrow;\, [\neg c_{1e}];\, p_e\downarrow;\, [\neg p_0];\, c_{10}\downarrow;\, [c_{1e}];\, p_e\uparrow\\
\quad []\, p_1 \wedge s_1 \longrightarrow c_{11}\uparrow;\, [\neg c_{1e}];\, p_e\downarrow;\, [\neg p_1];\, c_{11}\downarrow;\, [c_{1e}];\, p_e\uparrow\\
\quad []\, \neg p_\phi \longrightarrow s_0\downarrow, s_1\downarrow;\\
\qquad c_{0\phi}\downarrow, c_{1\phi}\downarrow;\, [\neg c_{0e} \wedge \neg c_{1e}];\, p_e\downarrow\boxed{;}\,[p_\phi];\, p_e\uparrow;\\
\qquad [p_0 \longrightarrow s_0\uparrow;\, p_e\downarrow;\, [\neg p_0];\, c_{0\phi}\uparrow;\, [c_{0e}];p_e\uparrow\\
\qquad []\, p_1 \longrightarrow s_1\uparrow;\, p_e\downarrow;\, [\neg p_1];\, c_{1\phi}\uparrow;\, [c_{1e}];p_e\uparrow]]]
\end{array}$$

After reset, the process resumes at $\boxed{;}$.

To realize these five branches as five concurrent processes, we must preclude the the first four from starting immediately after the fifth process executes $s_{0,1}\uparrow$. We accomplish this by replacing $s_{0,1}$ in their guards with $c_{0,1\phi}$, which also indicate the selected child.

$$\begin{array}{l}
*[[p_0 \wedge c_{0\phi} \longrightarrow c_{00}\uparrow;\, [\neg c_{0e}];\, p_e\downarrow;\, [\neg p_0];\, c_{00}\downarrow;\, [c_{0e}];\, p_e\uparrow\\
\quad []\, p_1 \wedge c_{0\phi} \longrightarrow c_{01}\uparrow;\, [\neg c_{0e}];\, p_e\downarrow;\, [\neg p_1];\, c_{01}\downarrow;\, [c_{0e}];\, p_e\uparrow\\
\quad []\, p_0 \wedge c_{1\phi} \longrightarrow c_{10}\uparrow;\, [\neg c_{1e}];\, p_e\downarrow;\, [\neg p_0];\, c_{10}\downarrow;\, [c_{1e}];\, p_e\uparrow\\
\quad []\, p_1 \wedge c_{1\phi} \longrightarrow c_{11}\uparrow;\, [\neg c_{1e}];\, p_e\downarrow;\, [\neg p_1];\, c_{11}\downarrow;\, [c_{1e}];\, p_e\uparrow\\
\quad []\, \neg p_\phi \longrightarrow s_0\downarrow, s_1\downarrow;\, ss\downarrow;\, v\downarrow;\\
\qquad c_{0\phi}\downarrow, c_{1\phi}\downarrow;\, [\neg c_{0e} \wedge \neg c_{1e}];\, p_e\downarrow\boxed{;}\,[p_\phi];\, p_e\uparrow;\\
\qquad [p_0 \longrightarrow s_0\uparrow;\, ss\uparrow;\, p_e\downarrow;\, [\neg p_0];\, v\uparrow;\, c_{0\phi}\uparrow;\, [c_{0e}];p_e\uparrow\\
\qquad []\, p_1 \longrightarrow s_1\uparrow;\, ss\uparrow;\, p_e\downarrow;\, [\neg p_1];\, v\uparrow;\, c_{1\phi}\uparrow;\, [c_{1e}];p_e\uparrow]]]]
\end{array}$$

$v$ is introduced to allow $c_{0,1\phi}$ to be combinational and $ss$ is introduced to reduce the length of $p_e$'s pull-up and pull-down chains (see PRS below).

The following PRS implements the receiver node's HSE.

$$\begin{array}{lcl}
p_\phi \wedge \neg ss \vee c_{0e} \vee c_{1e} & \rightarrow & p_e\uparrow\\
(\neg p_\phi \vee ss) \wedge \neg c_{0e} \wedge \neg c_{1e} & \rightarrow & p_e\downarrow
\end{array}$$

$$\begin{array}{lcllcl}
s_0 \vee s_1 & \rightarrow & ss\uparrow & \neg s_0 \wedge \neg s_1 & \rightarrow & ss\downarrow\\
\\
p_0 \wedge \neg v & \rightarrow & s_0\uparrow & \neg p_\phi & \rightarrow & s_0\downarrow\\
p_1 \wedge \neg v & \rightarrow & s_1\uparrow & \neg p_\phi & \rightarrow & s_1\downarrow\\
\\
ss \wedge \neg p_0 \wedge \neg p_1 & \rightarrow & v\uparrow & \neg ss & \rightarrow & v\downarrow\\
\\
v \wedge s_0 & \rightarrow & c_{0\phi}\uparrow & \neg v \vee \neg s_0 & \rightarrow & c_{0\phi}\downarrow\\
v \wedge s_1 & \rightarrow & c_{1\phi}\uparrow & \neg v \vee \neg s_1 & \rightarrow & c_{1\phi}\downarrow\\
\\
p_0 \wedge c_{0\phi} & \rightarrow & c_{00}\uparrow & \neg p_0 \vee \neg c_{0\phi} & \rightarrow & c_{00}\downarrow\\
p_1 \wedge c_{0\phi} & \rightarrow & c_{01}\uparrow & \neg p_1 \vee \neg c_{0\phi} & \rightarrow & c_{01}\downarrow\\
p_0 \wedge c_{1\phi} & \rightarrow & c_{10}\uparrow & \neg p_0 \vee \neg c_{1\phi} & \rightarrow & c_{10}\downarrow\\
p_1 \wedge c_{1\phi} & \rightarrow & c_{11}\uparrow & \neg p_1 \vee \neg c_{1\phi} & \rightarrow & c_{11}\downarrow
\end{array}$$

## V. Router Application

We connected a 2D array of spiking-neuron clusters to a datapath using our asynchronous serial tree-router, a natural choice for spike communication. A product of continuous, noisy analog dynamics at biological timescales, spikes are relatively infrequent (sub-kHz) and asynchronous (there's no clock). Each cluster contains 16 spike-generating soma circuits, 4 spike-consuming synapse circuits, and a configuration memory.

Clusters are tiled in a 16×16 array. To service the 4,096 somas, 1,024 synapses, and 256 memories, the transmitter's and receiver's trees are six ($4^6 = 4096$) and five ($4^5 = 1024$) levels deep, respectively. Half of the receiver's 1,024 output ports suffice to service all 1,024 synapses because each port supplies 2 bits (a 1-of-4 code) whereas each synapse needs only 1 bit (indicates whether a spike is excitatory or inhibitory). Thus, 512 ports are left over to service the 256 memories. We customized the transmitter's and receiver's leaf nodes to suit this application as follows.

### A. Transmitter Leaf

The transmitter leaf transmits a soma's spike up the tree by creating a packet containing the soma's index. With no other data to convey, the transmitter node is simplified to

$$\begin{array}{l}
\mathrm{TXL}(2) \equiv *[[\overline{C_0} \longrightarrow C_0\bullet(P;P!0);\, [\overline{C_0} \longrightarrow C_0 \bullet P]\\
\qquad\qquad\quad |\, \overline{C_1} \longrightarrow C_1\bullet(P;P!1);\, [\overline{C_1} \longrightarrow C_1 \bullet P]]
\end{array}$$

Note that our design actually instantiates $\mathrm{TXL}(4)$. We omit its HSE and PRS for brevity.

Somas lock up the transmitter during spike emission. When emitting a spike, a soma initiates packet transmission with a two-phase handshake ($C_{0,1}$). Afterwards, the soma enters a refractory period for up to a few milliseconds before executing another two-phase handshake ($C_{0,1}$) to terminate transmission. If communications within the transmitter are slackless, the soma will lock up the transmitter during its refractory period. To prevent this, we insert a buffer (i.e., latch) between the soma and the transmitter's leaf.

### B. Receiver Leaf

The receiver leaf services four synapses as well as a configuration memory (via a deserializer). We repurpose two of the receiver node's 2-bit ports to service the four synapses and use a third port to communicate with the memory:

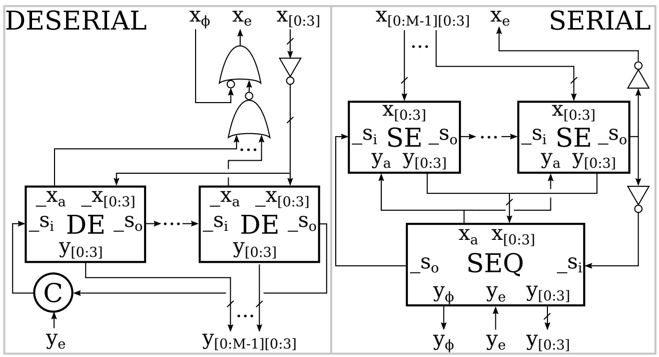

Fig. 5. Serial–Parallel Conversion. DESERIAL: Serial input fans out to a chain of $M$ DEs. An event moves from one to the next with each serial input; it loops back around through the C-element when parallel output occurs. SERIAL: Parallel input is divided among a chain of $M$ SEs. As an event moves from one to the next, it outputs its data to SEQ. The event loops back around through SEQ when serial transmission is complete.

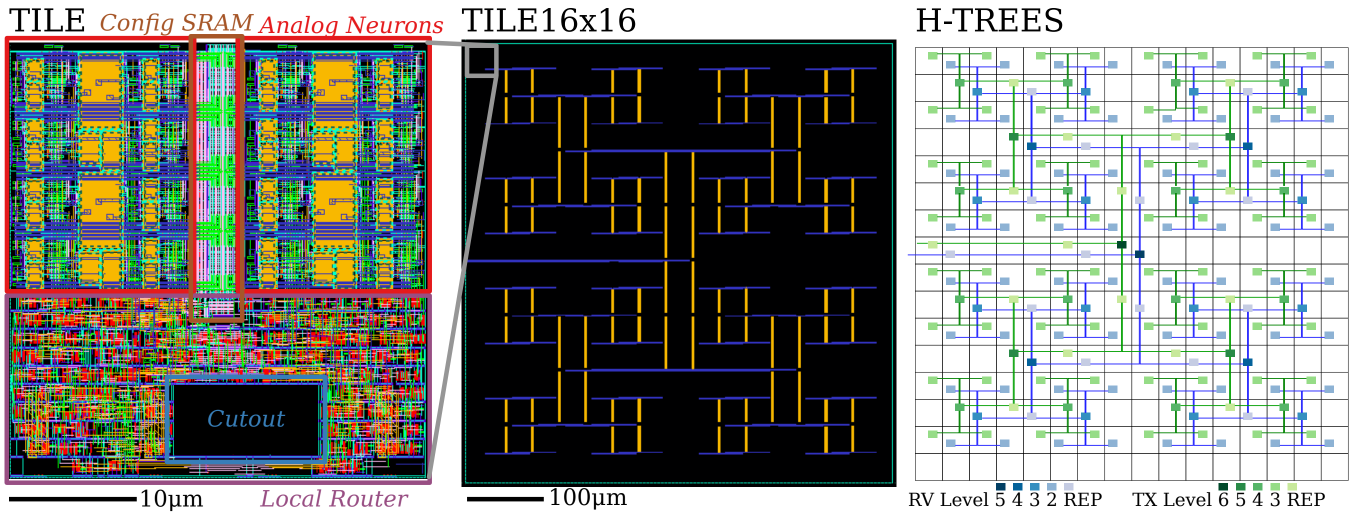

Fig. 6. Tree Router Layout. TILE: Neuron cluster and low-level, local router circuits. *Analog Neurons*: Circuitry for 16 somas and 4 synapses. *Config SRAM*: Sixty-four 2-bit words—tiled in 8 rows and 8 columns—for analog circuitry configuration. *Local Router*: Four transmitter-tree leaves, their parent, a receiver-tree leaf, and a deserializer (SRAM interface). *Cutout*: Populated as needed with the transmitter or receiver trees' higher-level nodes or repeaters. TILE16×16: Full 2D array. Digital signals enter and exit on its left side, where the datapath is attached. To minimize crosstalk with analog circuitry, H-tree wires runs over the TILEs' *Local Routers* and *Config SRAMs*. A one-tile horizontal displacement between the two H-trees makes wiring possible with just metal layers 5 (yellow) and 6 (purple). H-TREES: Placement of two H-trees' higher-level nodes and repeaters (green for transmitter and blue for receiver) in tiles (white squares) with routing overlaid. A vertical offset between the H-trees has been added for visual clarity only. TX: Transmitter; RV: Receiver; REP: Repeater.

$$
\begin{array}{l}
\mathrm{RVL}(4) \equiv P; P?s \bullet [s=2 \longrightarrow C_2 \,[\!]\, s \neq 2 \longrightarrow \mathrm{skip}];\\
\quad *[[\overline{P?} \wedge s=0 \longrightarrow C_0!P?\\
\quad\;\; [\!]\, \overline{P?} \wedge s=1 \longrightarrow C_1!P?\\
\quad\;\; [\!]\, \overline{P?} \wedge s=2 \longrightarrow C_2!P?\\
\quad\;\; [\!]\, \overline{P} \longrightarrow P\bullet [s=2 \longrightarrow C_2 \,[\!]\, s \neq 2 \longrightarrow \mathrm{skip}];\\
\quad\quad\;\; P; P?s \bullet [s=2 \longrightarrow C_2 \,[\!]\, s \neq 2 \longrightarrow \mathrm{skip}]]]
\end{array}
$$

Only the third port ($C_2$) continues with the serial protocol. HSE and PRS are omitted for brevity.

Synapses, like somas, require buffering. Depending on its analog biasing, a synapse may take up to a few milliseconds to acknowledge an input spike. We add a full cycle of slack to the otherwise slackless communication from root to synapse. One half-cycle is built into the leaf; a standard weak-precharge half-buffer provides the other.

The configuration memory accepts 6 bits of address and 2 bits of data (its 128 bits are organized into eight rows and eight 2-bit-wide columns). These 8 bits are encoded in four 1-of-4 codes that the deserializer receives in series from the receiver leaf and presents in parallel to the memory.

### *C. Serial–Parallel Conversion*

The deserializer converts $M$ sequentially delivered 1-of-4 codes into a $M$×1-of-4 parallel code using a chain of $M$ DEs (Fig. 5, DESERIAL). For 1-of-2 codes, DE's HSE is:

$$
\begin{array}{l}
*[[s_i]; [x_0 \longrightarrow y_0\uparrow; x_a\uparrow; [\neg x_0]; s_o\uparrow; x_a\downarrow; [\neg s_i]; y_0\downarrow; s_o\downarrow\\
\qquad\quad [\!]\, x_1 \longrightarrow y_1\uparrow; x_a\uparrow; [\neg x_1]; s_o\uparrow; x_a\downarrow; [\neg s_i]; y_1\downarrow; s_o\downarrow]]
\end{array}
$$

For each conversion, $s_{i,o}$ propagate an event along the chain twice. The first time, serial input codes are latched to build the parallel output. The second time, the parallel output is cleared, which happens once a C-element that closes the chain receives the environment's acknowledge. PRS for a 1-of-2 version of DE is as follows.

$$
\begin{array}{lll}
\neg s_o \wedge s_i \wedge x_0 & \rightarrow & y_0\uparrow\\
\neg s_o \wedge s_i \wedge x_1 & \rightarrow & y_1\uparrow\\
\neg s_o \wedge v_y & \rightarrow & x_a\uparrow\\
y_0 \vee y_1 & \rightarrow & v_y\uparrow\\
v_y \wedge \neg x_0 \wedge \neg x_1 & \rightarrow & s_o\uparrow
\end{array}
\qquad
\begin{array}{lll}
\neg s_i & \rightarrow & y_0\downarrow\\
s_o \vee \neg v_y & \rightarrow & x_a\downarrow\\
\neg s_i & \rightarrow & y_1\downarrow\\
\neg y_0 \wedge \neg y_1 & \rightarrow & v_y\downarrow\\
\neg v_y & \rightarrow & s_o\downarrow
\end{array}
$$

$v_y$ is introduced to shorten transistor chains.

The serializer does the converse of the deserializer: It uses a chain of $M$ SEs to slice a $M$×1-of-4 parallel code into $M$ 1-of-4 codes and forwards them sequentially to the environment using SEQ (Fig. 5, SERIAL). For 1-of-2 codes, SE's HSE is:

$$
\begin{array}{l}
*[[s_i];\\
[x_0 \longrightarrow y_0\uparrow; [y_a]; u\uparrow; y_0\downarrow; [\neg y_a]; s_o\uparrow; [\neg s_i]; u\downarrow; [\neg x_0]; s_o\downarrow\\
[\!]\, x_1 \longrightarrow y_1\uparrow; [y_a]; u\uparrow; y_1\downarrow; [\neg y_a]; s_o\uparrow; [\neg s_i]; u\downarrow; [\neg x_1]; s_o\downarrow\\
]]
\end{array}
$$

$u$ is added to distinguish states before and after the $y_{0,1}$-$y_a$ handshake. As with the deserializer, $s_{i,o}$ propagate an event along the chain twice for each conversion. The first time, each SE relays a code to SEQ. The second time, each SE checks that its parallel-input slice is cleared. SE's PRS is:

$$
\begin{array}{lll}
x_0 \wedge \neg u \wedge s_i & \rightarrow & y_0\uparrow\\
x_1 \wedge \neg u \wedge s_i & \rightarrow & y_1\uparrow\\
s_i \wedge y_a & \rightarrow & u\uparrow\\
u \wedge \neg y_a & \rightarrow & s_o\uparrow
\end{array}
\qquad
\begin{array}{lll}
u \wedge \neg s_o & \rightarrow & y_0\downarrow\\
u \wedge \neg s_o & \rightarrow & y_1\downarrow\\
\neg s_i & \rightarrow & u\downarrow\\
\neg u \wedge \neg x_0 \wedge \neg x_1 & \rightarrow & s_o\downarrow
\end{array}
$$

SEQ's HSE is:

$$
\begin{array}{l}
*[[s_i]; s_o\uparrow; [x_0 \vee x_1]; y_\phi\uparrow; [\neg s_i \wedge y_e]; y_\phi\downarrow; [\neg y_e]; s_o\downarrow],\\
*[[x_0 \wedge y_e \longrightarrow y_0\uparrow; [\neg y_e]; x_a\uparrow; [\neg x_0]; y_0\downarrow; [y_e]; x_a\downarrow\\
\quad [\!]\, x_1 \wedge y_e \longrightarrow y_1\uparrow; [\neg y_e]; x_a\uparrow; [\neg x_1]; y_1\downarrow; [y_e]; x_a\downarrow]]
\end{array}
$$

By closing the chain, it initiates ($y_\phi\uparrow; [y_e]$) and terminates ($y_\phi\downarrow; [\neg y_e]$) packet transmission on the event's first and second pass, respectively. In between, it forwards codes that SEs provide. SEQ's PRS is:

$$\begin{array}{lll@{\qquad}lll}
x_0 \vee x_1 & \to & y_\phi\uparrow & \neg s_i \wedge y_e & \to & y_\phi\downarrow \\
(y_0 \vee y_1) \wedge \neg y_e & \to & x_a\uparrow & y_e & \to & x_a\downarrow \\
y_e \wedge x_0 & \to & y_0\uparrow & \neg x_0 & \to & y_0\downarrow \\
y_e \wedge x_1 & \to & y_1\uparrow & \neg x_1 & \to & y_1\downarrow \\
\neg s_i \wedge \neg y_e \wedge \neg y_\phi & \to & s_o\downarrow & s_i & \to & s_o\uparrow
\end{array}$$

## VI. Synthesis and Validation

For logical synthesis, we described logical hierarchy and PRS in the Asynchronous Compiler Tools (ACT) language.[4] We verified logical correctness with PRSIM, a discrete-event simulator that executes PRS with randomized delays [13], and then checked for logical-physical consistency in the presence of transistor parasitic capacitances with CoSIM, a PRSIM–SPICE co-simulator [13].

For physical synthesis, we decomposed our ACT into standard cells and generated their layouts with cellTK [14]. Encounter (*Cadence*) place-and-routed lower-level router circuitry—4 transmitter leaves (to service 16 somas), their parent node, a receiver leaf (to service 4 synapses and an SRAM), and a deserializer (to interface with the SRAM)—in the lower 43% ($547\mu\text{m}^2$) of the neuron-cluster tile ($1,261\mu\text{m}^2$). Of this router area, 14% ($76\mu\text{m}^2$) was reserved (cutout) for higher-level circuitry (Fig. 6, TILE).

We placed tiles in a 16×16-array and placed the router's higher-level nodes in their cutouts, along with repeaters to drive long wires (Fig. 6, TILE16×16, and H-TREES). We extracted parasitic resistances and capacitances from this layout and performed simulations to check for spurious transitions and to predict the router's maximum throughput.

Our postlayout simulations predicted that the transmitter and receiver could communicate up to 42.5 and 50.8 Mspike/s, respectively (Fig. 7).[5] Codes from (or to) nodes lower in the tree take longer (e.g., 4.31 ns from the transmitter tree's leaves versus 1.28 ns from its root) because the number of communications involved increases (from 6 at the leaf to 1 at the root). On average, 4.5 four-phase communications are performed, including one for the 2 two-phase communications that demarcate the packet. At 4 phases per communication and 4 transitions per phase, transversing six nodes involves 432 transitions.[6] Thus, the 42.5 Mspike/s cycle-rate corresponds to 56 ps per transition, in line with expectations for a 28-nm process.

Post-fabrication in a 28-nm, fully depleted, silicon-on-insulator process, we brought the chip up and validated the router's functionality. (Fig. 8). From two chips, we measured maximum throughputs of 27.4 and 26.1 Mspikes/s for the transmitter and 18.1 and 18.5 Mspikes/s for the receiver. Differences between simulations the chip measurements are explained by additional delays introduced by unpipelined datapath communications.

[4]Available at https://github.com/samfok/AER_serial_tree_router

[5]In operation, somas generate up to 500 spike/s each, and synapses consume up to 1000 spikes/s each, so we expect the transmitter and receiver to communicate 2 and 1 Mspikes/s, respectively

[6]In our PRSIM simulations, we counted 422 transitions for the transmitter and serializer and 481 transitions for the deserializer and receiver.

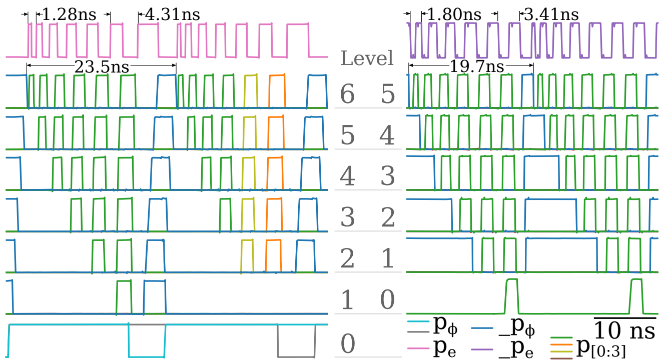

Fig. 7. Postlayout Transmitter and Receiver SPICE Simulations. *Left*: SOMA 0 and 9 (000000 and 000021 in 4-ary) spike simultaneously (Level 0). Their $p_\phi$ signals propagate up to Level 1 (only SOMA 0's parent is shown) and Level 2, where SOMA 0's subtree is selected. Thus, its $p_\phi$ signal propagates to the root (Level 6). Enabled by the environment's $p_e$ signal (top), which propagates down the tree (not shown), each node forwards its requesting child's 1-of-4 coded index ($p_{0:3}$) and then forwards indices forwarded by the child (they are all 0 for SOMA 0). Each node clears its $\_p_\phi$ once its child clears its $\_p_\phi$, signaling that there are no more indices to be forwarded. A node is then free to select another requesting subtree, as happens at Level 2 for SOMA 9's subtree. *Right*: The environment sends two inhibitory spikes to SYNAPSE 0 by injecting two packets containing its path appended with 0 (i.e., 000000) at the root. Each node selects the child indexed by the first 1-of-4 code ($p_{0:3}$) and forwards the remaining codes to that child after lowering $\_p_\phi$. Note that the leaf (Level 1) does not propagate $\_p_\phi$ to the synapse (Level 0).

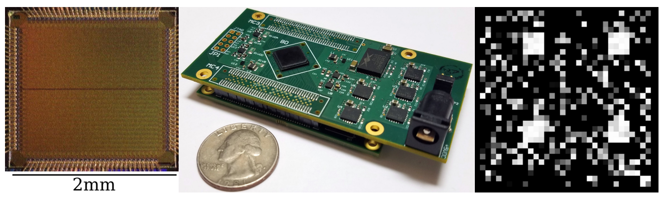

Fig. 8. Fabrication and validation. *Left*: Test chip containing analog neurons, the router presented herein, and a digital datapath. *Center*: Test board piggybacked on an FPGA development board (Opal Kelly) that provides a USB link to a host computer. *Right*: Visualization of a 32×32-soma patch of the chip's spiking activity. Each small square represents a soma; its brightness reflects the soma's spike rate. The four, bright soma clusters are receiving excitatory input from spikes delivered to nearby synapses.

## VII. Discussion

Pioneering researchers developed transmitters and receivers to write and read spikes to and from 1D or 2D arrays of silicon neurons using the *address-event representation* (AER; [4], [5], [15], [16]). A neuron's address is transmitted every time it spikes, hence the name *address-event*. In 1D, the spike is identified by a unique address assigned to each neuron. In 2D, the spike is identified by the neuron's row and column addresses and, in first-generation designs, these addresses are transmitted in parallel.

Second-generation designs communicated row and column addresses in series. In addition to saving wires by multiplexing, this so-called *word-serial* protocol supports packets with an arbitrary number of words. Thus, additional column addresses could be appended to communicate multiple spikes read from or written to the same row in parallel [6]–[8]. This so-called *burst-mode* offered higher throughput, servicing arrays containing as many as 64k somas and 256k synapses [1] at rates up to 43.4M spike/s (ignoring off-chip delays) [17]. Array or chip addresses could be prepended to further expand the address-space. Thus, an address-event-based router could service multiple arrays distributed across multiple chips [18]–[20]. Further, data as well as addresses could be communicated over the link (or bus) connecting the transmitter to the receiver [21]. In this fashion, multiple spikes read in parallel from small groups of neurons have been transmitted using a single dataword, boosting throughput, which had plateaued at 50M spike/s [22], to 300M spike/s [23].

To communicate configuration datawords to or from individual neurons—or clusters thereof—we could widen the neuronal interface, but the additional bandwidth would be largely wasted. Datawords use all of the wires but occur rarely, whereas spikes occur frequently but only use one (e.g., a soma's output) or two (e.g., a synapse's excitatory or inhibitory input) wires. We thus keep the neuronal interface narrow and transmit data serially, saving wires by taking more time. In addition to supporting multiple data-types efficiently, a serial protocol places no limit on the number of bits a dataword can have, unlike a parallel protocol.

We did away with timing-assumptions by switching from row-column addresses to tree paths. Striking a balance between node-count and node-complexity, we chose a 4-ary over a binary tree, which reduced transistor-count by 19.1% overall. Returns diminish for higher degrees because realizing wider gates requires treeing narrower gates (with no more than four transistors in series).[7] Although its thin-oxide transistors outnumber the neuron-cluster's thick-oxide transistors 1.9:1, the router takes up only 43% of the total area because thick-oxide transistors are much larger than thin oxide transistors.

Throughput may be enhanced substantially by pipelining the otherwise slackless communication from leaf to root (transmitter) or root to leaf (receiver) and between router and datapath. Pipelining can be added to the current design at no additional area cost by replacing repeaters with latches or placing latches in unused tile cutouts. Subsequent codes would take no more time than the first one, which takes 1.28 (transmitter) or 1.8 ns (receiver) (Fig. 7). Therefore, with seven communications per spike, pipelining would increase throughput from 42.5M to 111.6M spike/s (transmitter) or from 50.7M to 79.4M spike/s (receiver).

## Acknowledgements

The authors thank Prof. R. Manohar, Computer Systems Lab, Yale University, for guidance on asynchronous circuit design, as well as CAD tools for their synthesis and verification. We thank our colleagues at the Brains in Silicon Lab of Stanford University for stimulating discussions. This project was made possible by grants from the Office of Naval Research (N000141310419 and N000141512827).

[7]Gates with longer chains operate much slower, increasing the duration that downstream gates pass short-circuit currents.

## References

[1] B. V. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[2] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[3] J. Park *et al.*, "Hierarchical address event routing for reconfigurable large-scale neuromorphic systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2408–2422, 2017.

[4] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address-events," *IEEE Transactions on Circuits & Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 1999.

[5] M. Mahowald, *An analog VLSI stereoscopic vision system*, vol. 1. 1994.

[6] K. A. Boahen, "A burst-mode word-serial address-event link-I: transmitter design," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 7, pp. 1269–1280, 2004.

[7] K. A. Boahen, "A burst-mode word-serial address-event link-II: receiver design," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 7, 2004.

[8] J. Lin and K. Boahen, "A delay-insensitive address-event link," in *IEEE Symposium on Asynchronous Circuits and Systems*, 2009.

[9] K. Boahen, "A neuromorph's prospectus," *Computing in Science and Engineering*, vol. 19, no. 2, pp. 14–28, 2017.

[10] A. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, MIT Press, 1990.

[11] R. Manohar and Y. Moses, "The eventual C-element theorem for delay-insensitive asynchronous circuits," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 102–109, 2017.

[12] A. J. Martin and M. Nyström, "Asynchronous techniques for system-on-chip design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, 2006.

[13] F. Akopyan, C. Tadeo, O. Otero, and R. Manohar, "Hybrid synchronous-asynchronous tool flow for emerging VLSI design," in *IEEE International Workshop on Logic Synthesis*, 2016.

[14] R. Karmazin, C. T. O. Otero, and R. Manohar, "cellTK: automated layout for asynchronous circuits with nonstandard cells," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, 2013.

[15] M. A. Sivilotti, *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. PhD thesis, Pasadena, CA, USA, 1991.

[16] J. Lazzaro *et al.*, "Silicon auditory processors as computer peripherals," *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 523–528, 1993.

[17] K. A. Boahen, "A burst-mode word-serial address-event link-III: testing and results," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 7, pp. 1292–1300, 2004.

[18] T. Y. W. Choi *et al.*, "Neuromorphic implementation of orientation hypercolumns," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52-I, pp. 1049–1060, 2005.

[19] P. A. Merolla *et al.*, "Expandable networks for neuromorphic chips," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 2, pp. 301–311, 2007.

[20] P. Merolla *et al.*, "A multicast tree router for multichip neuromorphic systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 3, pp. 820–833, 2014.

[21] J. Georgiou, A. G. Andreou, and P. . Pouliquen, "A mixed analog/digital asynchronous processor for cortical computations in 3D SQl-CMOS," in *2006 IEEE International Symposium on Circuits and Systems*, p. 4, 2006.

[22] C. Brandli *et al.*, "A 240 180 130 dB 3 $\mu$s latency global shutter spatiotemporal vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 10, pp. 2333–2341, 2014.

[23] B. Son *et al.*, "A 640480 dynamic vision sensor with a 9m pixel and 300Meps address-event representation," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, pp. 66–67, 2017.