A Multicast Tree Router For Multichip Neuromorphic Systems

Paul Merolla, John Arthur, Rodrigo Alvarez, Jean-Marie Bussat, Kwabena Boahen

***Abstract*—We present a tree router for multichip systems that guarantees deadlock-free multicast packet routing without dropping packets or restricting their length. Multicast routing is required to efficiently connect massively parallel systems' computational units when each unit is connected to thousands of others residing on multiple chips, which is the case in neuromorphic systems. Our tree router implements this one-to-many routing by branching recursively—broadcasting the packet within a specified subtree. Within this subtree, the packet is only accepted by chips that have been programmed to do so. This approach boosts throughput because memory look-ups are avoided enroute, and keeps the header compact because it only specifies the route to the subtree's root. Deadlock is avoided by routing in two phases—an upward phase and a downward phase—and by restricting branching to the downward phase. This design is the first fully implemented wormhole router with packet-branching that can never deadlock. The design's effectiveness is demonstrated in *Neurogrid*, a million-neuron neuromorphic system consisting of sixteen chips. Each chip has a 256×256 silicon-neuron array integrated with a full-custom asynchronous VLSI implementation of the router that delivers up to 1.17G words/s across the sixteen-chip network with less than 1μs jitter.**

***Index Terms*—Neuromorphic, Asynchronous, VLSI, router, multicast, deadlock, tree network.**

I. Neuromorphic Networks

Neuromorphic chips must be interconnected by routing networks to match the scale of the brain, which performs powerful and energy-efficient computation with billions of slow and noisy neurons, each reconfigurably connected to thousands of others. Slow and noisy neurons are emulated with subthreshold analog circuits [1] that consume similar amounts of current as in biology—picoamps to nanoamps—computing with devices that are essentially off (i.e., dark silicon [2]). Recent work has explored digital neuron implementations [3, 4]. Massive and reconfigurable connectivity is implemented with fast, time-multiplexed, asynchronous digital circuits that communicate the spikes these silicon neurons emit [5, 6]. They are often organized in two-dimensional (2D) arrays and serviced by a transceiver that reads spikes from and writes spikes to a row in parallel [7, 8], an embedded memory that provides reconfigurable connectivity [9, 10], and an on-chip router that communicates spike packets between chips [11, 12] (Fig. 1a).

The NIH Director's Pioneer Award provided funding (DPI-OD000965)

K. Boahen is with the Department of Bioengineering, Stanford University, Stanford, CA, 94305 USA. All the other authors were with Stanford when the work was performed. Correspondence should be directed to K. Boahen at boahen@stanford.edu and P. Merolla at pmerolla@gmail.com

Copyright (c) 2013 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

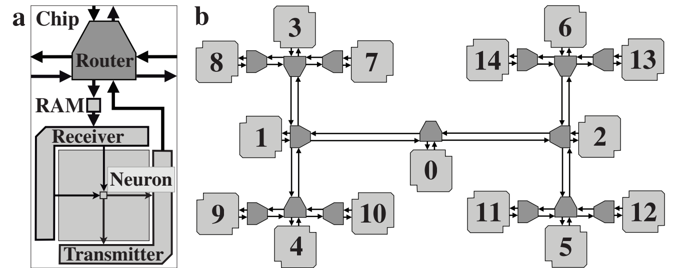

Fig. 1. Multichip Neuromorphic System. **a** Neuromorphic chip: Integrates a router, a silicon neuron array, a receiver and a transmitter that communicate spikes to and from the array, a RAM that provides flexible connectivity, and an on-chip router. **b** Fifteen-node binary tree: Each node is a chip. On-chip routers, connected by interchip links, service their own array's spikes and relay other arrays' spikes up and down the tree.

Routers for neuromorphic systems are designed to satisfy four requirements: First, because a packet can be very long—carrying all the spikes from a neural array's row (see [7])—the router should not wait to receive the entire packet before forwarding it to the next chip. The alternative requires the router to buffer the entire packet locally, which is prohibitive in terms of silicon area. Second, because a neural array consumes packets sequentially—writing all of a packet's spikes to its target row in parallel (see [8])—the router can not interleave packets. Third, because each silicon neuron is potentially connected to thousands of others, the router should support multicast routing to efficiently deliver spikes to several chips.[1] Fourth, because spike times are used to encode information [14–20], the router should have extremely low latency.

Meshes have been proposed for connecting neuromorphic chips together in a scalable fashion to create large-scale networks [21, 22] because they offer a large channel bisection[2] ($O(\sqrt{n})$ for n nodes). However, meshes have long latencies, due to their large diameter[3] ($O(\sqrt{n})$). They also do not guarantee deadlock-free multicast routing (i.e., one to many) [23], which introduces additional packet dependencies [24] (see Section II). When these dependencies form a closed cycle, packets do not make progress towards their destinations; hence the routing network is said to be deadlocked. To avoid long

[1]In addition to multicast routing, one-to-many connectivity can also be implemented within the neural array via tunable analog diffusors that model a resistive sheet [13], specialized receiver circuitry (see [10]), or local crossbar memory (see [4]). However, these fanout techniques are not the focus of our paper.

[2]Channel bisection is defined as the minimum (worst case) number of links connecting two halves of the network across all possible bisections [23].

[3]Diameter is defined as the largest minimal path, in number of links transversed, between any pair of nodes [23].

TABLE I
MESH VERSUS TREE FOR ALL-TO-ALL TRAFFIC

Packets relayed	Mesh Unicast	Tree Unicast	Tree Multicast
Average	$\mathrm{O}(n^{3/2})$	$\mathrm{O}(n\log(n))$	$\mathrm{O}(n)$
Peak	$\mathrm{O}(n^{3/2})$	$\mathrm{O}(n^2)$	$\mathrm{O}(n)$

latencies and guarantee deadlock-free multicast routing, we revisited the tree topology (Fig. 1b), which offers short latency ($\mathrm{O}(\log n)$ diameter) and is provably deadlock-free for point-to-point (i.e., unicast) routing performed in two phases—an upward phase and a downward phase [25–27].

In this paper, we implement deadlock-free multicast routing in a tree network for the first time (to the best of our knowledge). We use up-down, point-to-point routing to target a particular subtree and recursive branching to broadcast within that subtree. Deadlock is avoided by restricting branching to the downward phase (unlike in [28]). Routing one packet to multiple destinations—multicasting—in this fashion cuts bandwidth requirements by a factor equal to the fanout, relieving the bottleneck at the tree's root ($\mathrm{O}(1)$ channel bisection). The peak number of packets the root relays for all-to-all traffic ($\mathrm{O}(n^2)$ messages) drops from $\mathrm{O}(n^2)$ for unicast to $\mathrm{O}(n)$ for multicast.[4] In contrast, each node relays $\mathrm{O}(n^{3/2})$ packets in the mesh ($\mathrm{O}(\sqrt{n})$ channel bisection), which is restricted to unicast to avoid deadlock. Meshes are provably deadlock-free in the unicast case when dimension-order routing or virtual channels are used [23]. Table I provides a complete comparison; the tree's and the mesh's unicast scaling is derived in [27].

Section II describes the tree network's routing algorithm, which routes a packet from one to many chips. Section III describes the router's logical design, which is fully asynchronous, offering the advantage of consuming energy only when there is spike activity.[5] Section IV describes the router's physical design, which was full-custom, and its implementation in Neurogrid. Section V presents test results, showing a performance of over one billion words delivered per second with jitter under one microsecond. Section VI concludes the paper with a brief discussion.

II. ROUTING ALGORITHM

We present a routing algorithm for a binary-tree network that supports local broadcast with branching while guaranteeing deadlock-free operation. A network consists of nodes connected by links. Links carry packets that start with a headword, followed by an arbitrary number of payload words, and end with a tailword. A link can only carry one word at a time, and commits to sending all the words of a particular packet before servicing another packet. It sends each word ahead as soon as possible—a protocol commonly known as wormhole routing [23]. This protocol is necessary since packets can be arbitrarily long, and therefore a node may not have the capacity to store the entire packet. So a packet may occupy several nodes at the same time, hence the worm analogy.

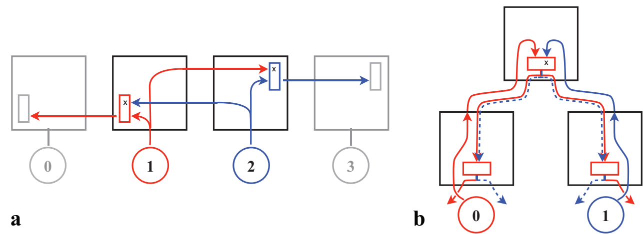

Fig. 2. Wormhole Routing Deadlocks When Branching Precedes Merging. **a** Mesh: Node 1's packet (red route) acquires the merge (copies packets from two incoming paths to one outgoing path) on its left but fails to acquire the merge on its right; the situation is reversed for Node 2 (blue route). Since each packet must acquire both merges to proceed, progress stalls. **b** Tree: Node 0's packet (red route) acquires the merge in the parent of its destination nodes, gaining unencumbered access to those nodes. Node 1's packet (blue route) waits at this merge until Node 0's packet is delivered, and then proceeds as well. Notice that deadlock is avoided by reversing the order in which branching and merging occur.

With wormhole routing, deadlock occurs in meshes because of the additional packet dependencies branching creates (Fig. 2a).[6] Deadlock has been dealt with by interleaving words from up to n packets on the same link (for n nodes) [30], extending the approach used in the unicast case, where two packets are interleaved using virtual channels on the same physical link [31]. In both cases, the destination must either provide enough buffering to locally reconstruct entire packets or employ non-blocking multiport memory structures to do so (see Discussion in [30]). These options are not viable for our application. First, packets are arbitrarily long, making buffering infeasible. Second, packets can branch recursively, requiring an arbitrarily large number of memory ports (up to n). Therefore, neither of these solutions are applicable.

A. *Local broadcast in a tree*

Our solution for implementing branching preserves the deadlock-free property of up-down routing in a tree (Fig. 2b). A tree is naturally deadlock-free for unicast communication (i.e., point-to-point), where a packet is routed *up* the tree to an intermediate node (the ancestor). The ancestor merges all packets destined for its descendants, giving the chosen packet exclusive access when it is routed *down* to its destination. In other words, while upward progress depends on downward progress, downward progress does not depend on upward progress, eliminating any possible cyclic dependency—a necessary condition for deadlock. Branching can be supported provided that it is restricted to the down phase. This restriction is necessary because if an upward traveling packet branches—is copied from a node's daughter to its parent as well as to its other daughter—this action will cause downward progress to depend on upward progress (and vise versa), thereby introducing a cycle.

[4] In the unicast case, the root's two daughters actually carry 25% more traffic, on average, than the root does [27].

[5] An additional benefit is reduced noise coupling into sensitive analog circuits, which model neurons and synapses more compactly and efficiently than digital circuits [29].

[6] Although path-based schemes avoid branching [24], deadlock can still occur when the packet is copied to destinations enroute, because multicast packets may contend for the same destinations.

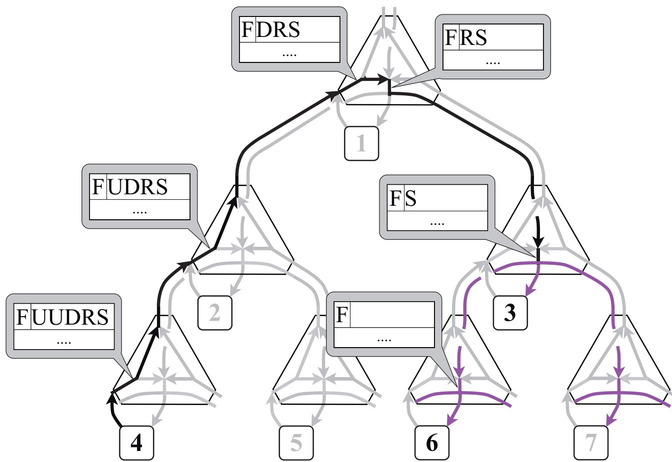

Fig. 3. Multicast Routing's Point-to-Point and Branching Phases. In this example, $A = 4$, $B = \{3, 6\}$, $P_{AB} = 1$, and $P_B = 3$. In the point-to-point phase (black), Node 4's packet is routed up to Node 1, an ancestor shared with its destination nodes (3 and 6). The packet is then routed down to Node 3, the destinations' ancestor. At each node, the route field's most-significant bit—shifted out—encodes which of two turns to take (U or D, R or L); a stop code (S) encodes the terminus (Node 3). In the branching phase (purple), the packet visits Node 3 and all its descendants (i.e., *floods*). A mode bit (F) determines whether the packet floods or targets the terminus. Notice that, since a node and its router reside on the same chip, the node's neural array has direct access to its router. Thus, spikes may be routed among its silicon neurons without leaving that chip (the route is simply D followed by S).

We use a version of up-down routing that consists of a point-to-point phase and branching phase, whereby a packet is sent from a source A to multiple destinations B. First, in the point-to-point phase, the packet travels up the tree until it reaches the lowest common ancestor of the source and all the destinations P_{AB}. At this ancestor node, the packet reverses direction and travels down the tree until it reaches the lowest common ancestor of all the destinations P_B. Second, in the branching phase, the packet branches recursively—each node copies the packet from its parent to its two daughters—visiting all nodes in that subtree (i.e., floods). Nodes that are not destinations filter packets using information stored at that node, achieving high-throughput by avoiding memory lookups while branching.[7]

B. Packet routing specifics

The point-to-point route from A to P_B is encoded in the packet's headword using a relative-addressing scheme that is tailored for traversing a binary tree (Fig. 3). The succession of ups (U) followed by a down (D) from A up to P_{AB} and right–left turns (R or L) from P_{AB} down to P_B are each encoded as `1` or `0`, respectively, terminated by a stop code (S)—uniquely identified as `1` followed by `0`s.[8] At each node, the route field's most-significant bit (MSB) is shifted out to determine the turn and a `0` is shifted in at the least-significant bit (LSB). In the up-phase, the packet keeps moving up if a `1` is shifted out (encodes U) but turns and goes down if a `0` is shifted out (encodes D). Once in the down-phase, a `0` or a `1` determines if the packet turns either left (L) or right (R), respectively. The packet proceeds until the route field is `1` (MSB) followed by `0`'s—the stop code (S)—which signifies it has arrived at P_B. In Neurogrid, this encoding makes it possible to span a four-level tree (15 nodes) with the nine bits available in the packet's twelve-bit headword (three bits are reserved for other functions); supporting more than four-levels is possible, however, this would require a wider datapath.

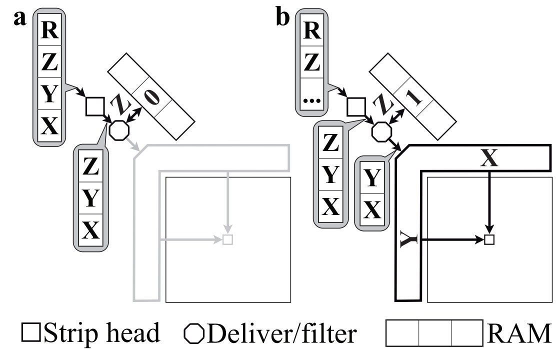

Fig. 4. Filtering and Delivering Packets to a Neural Array. **a** A spike packet contains a route (R) together with an address specifying the source neuron's array (Z), row (Y), and column (X). At the destination, R is stripped, and an on-chip RAM uses Z to look up a bit that indicates the packet's fate. In this case, it is filtered. **b** In this case, it is delivered. Note that the on-chip RAM does not perform address translation, which is implemented off-chip (see Section IV.C).

The additional step required to filter (i.e., reject) packets at nodes that are not in B is based on a scheme described in [9]. The packet's action at P_B is determined by an additional field in the headword: the *flood* bit (F). If the flood bit is set, the packet recursively branches to all P_B's descendants (flood mode).[9] Otherwise, it is delivered to P_B exclusively (target mode). In the former case, information stored in a local SRAM, accessed with an address provided in the packet's second word, is used to filter packets at nodes that are not in B. Depending on how the SRAM is programmed, the packet is either filtered or delivered (Fig. 4). If the packet is delivered, our current system appends two additional bits retrieved from the SRAM to its row address (third word); they specify one of four programmable synapse types to activate in the target neuron (e.g., fast or slow excitation, or fast or slow inhibition) [9].[10] Local fanout is implemented by tunable analog diffusors, which model a resistive sheet [13].

In addition to handling spikes, the router is designed to handle configuration and control packets, enabling the entire system to operate with a single communication fabric. We

[7] Memory lookups in our design, which are the slowest operation, do not impede the packet from being sent to another node. Therefore, all these lookups can proceed in parallel, in contrast to a network where the memory lookup decides the packet's route.

[8] Note that two bits (D followed by L or R) are used to turn around, although one should suffice since there are only two choices: continue going up or go out the other port. To use one bit, however, the hardware must keep track of the port the packet entered on, which adds complexity.

[9] More complex routing schemes are also possible whereby a branching packet only visits a subtrees n highest levels, but the hardware becomes significantly more complex. Furthermore, the gains are marginal as bandwidth becomes more plentiful as you go down the tree (i.e., away from the root and towards the leaves).

[10] In addition, a third bit provides four additional options that are used to disable a particular neuron or select its analog signals to be read out and digitized.

TABLE II
CHP LANGUAGE CONSTRUCTS

Operation	Notation	Explanation
Process	P_i	A composition of communications
Boolean	B_i	A boolean expression
Guarding	G_i	$\equiv B_i \rightarrow P_i$. Execute P_i if B_i is true
Sequential	$P_1; P_2$	P_1 ends before P_2 starts
Concurrent	$P_1 \| P_2$	$\|$ takes precedence over ;
Repetition	$*[P_1; P_2]$	$\equiv P_1; P_2; P_1; P_2; \ldots$. Repeats forever
Selection	$[G_1 \,[\!]\, G_2]$	Execute P_i for which B_i is true
Arbitration	$[G_1 \mid G_2]$	Required if B_i not mutually exclusive
Input	$A?x$	Read data from port A to register x
Output	$A!x$	Write data from register x to port A
Transfer	$B!(A?)$	Write data read from port A to port B
Probe	$\overline{A}$	Is communication pending on port A?
Index	$x.n$	nth bit or field n of x
Assign	$y := x$	Copy data from x to y
Concatenate	$y.\{i, j\}$	Concatenate $y.i$ and $y.j$

describe the different packet types and their format in the Appendix for the interested reader.

III. LOGICAL DESIGN

We implement our routing algorithm with quasi-delay insensitive (QDI) asynchronous circuits, following a synthesis method pioneered by Martin [32].[11] First, the design is specified in a high-level language, Communicating Hardware Processes (CHP, see Table II). A communication synchronizes the two processes communicating and may also transfer data between them. It is implemented in a delay-insensitive manner by fleshing it out into a request–acknowledge sequence, a process known as hand-shaking expansion (HSE). Finally, the HSE is decomposed into a set of boolean expressions (called guards) that enforce the specified order of signal transitions. The guards and their corresponding transitions, which are called a production rule set (PRS), are then directly implemented with transistors (refer to Appendix for synthesis examples).

A. CHP specification

The router process provides four functions: (1) Accepts packets originating from its local spike transmitter or analog-to-digital converter (ADC)[12]; (2) Relays packets traveling up the tree from either of its daughters to its parent; (3) Relays packets traveling down the tree from its parent to either or both of its daughters; (4) Delivers packets that have reached their destination to one of two local memories; one stores synaptic connectivity, the other stores neuron parameters (see Appendix for details). To accomplish these tasks, it communicates with the transmitter and ADC on two input ports (Tx and ADC), with the daughters on two bi-directional ports ($\mathrm{L_i}$,$\mathrm{L_o}$ and $\mathrm{R_i}$,$\mathrm{R_o}$), with the parent on a third bidirectional port ($\mathrm{T_i}$,$\mathrm{T_o}$), and with the local memories on two output ports (Rv and Bias). It can be neatly decomposed into two- and three-input merges (Merge and Merge3, respectively), that merge traffic onto the up and the down paths, and two kinds of splits (Up and Down), that make routing decisions on the up and down paths (Fig. 5).

Merge feeds packets on two incoming paths (input on L and R) into a single outgoing path (output on O) on a first-come-first-serve basis. Its CHP reads:

```
     Merge
≡    h := true;
     *[ [ h →
            [ ‾L → s := left; O!(L?)
            | ‾R → s := right; O!(R?)
            ]; h := false
          [] ¬h →
            [ s = left → L?x; O!x
            [] s = right → R?x; O!x
            ]; h := x.tail
      ] ]
```

On initialization, an internal variable h is set to true, indicating that the next word to be processed is a head; subsequent ones are processed as payload. The first clause, which is executed when a head is expected, arbitrates between L and R, updates s to reflect its choice, and transfers the headword from the selected input to the output. Then, h is set to false, indicating that subsequent words are payload. The second clause, which executes when a payload word is expected, transfers the rest of the packet to O, copying each word to x to capture the tail bit (used to update h). When the last word in the packet is reached, h is re-initialized to true, since x.tail is true, preparing the process for a new packet. We omit Merge3's CHP for economy. However, the code above can easily be extended to handle any number of inputs.

Up reads the route encoded in an incoming packet's head (input on A) and decides whether to keep it on the up path (output on U), direct it to the down path (output on D), or consume it (no output). Its CHP reads:

```
     Up
≡    h := true;
     *[ A?x;
          [ h →
              dir := x.route[msb] ||
              y.route := x.route × 2 ||
              y.{mode, mem} := x.{mode, mem};
                 [ y.route = 0 → s := stop
                 [] y.route ≠ 0 ∧ dir = u → s := up; U!y
                 [] y.route ≠ 0 ∧ dir = d → s := down; D!y
                 ]
          [] ¬h →
              [ s = stop → skip
              [] s = up → U!x
              [] s = down → D!x
              ]
          ]; h := x.tail
     ]
```

The first clause, which is executed only for the head, extracts the route field's MSB (stored in dir) and computes the new route (y.route) by multiplying the old one by 2 (shifting the bits toward the MSB). Based on dir and y.route, the packet

[11] The design methodology is *quasi* (as opposed to fully) delay-insensitive because some circuits require wire branches to have smaller delays than gates—otherwise a race condition can occur. In practice, it is relatively straightforward to ensure the physical design meets the required timing assumptions.

[12] The ADC can measure the internal analog values of a subset of signals for any neuron in the chip, and convert them into packets that are sent off chip.

has three route options (stop, up, or down);[13] the selected option, which is saved in s, also applies to subsequent payload words. After dispatching the head with the updated route to the appropriate output, or skipping this operation in the case of a stop, h is set to false (x.tail is false for all except the last word in the packet). The second clause, which is executed only for the payload, selects the appropriate action (skip, $U!x$, or $D!x$) based on s. When the last word in the packet is reached, h is re-initialized to true, since x.tail is true, preparing the process for a new packet.

We can further decompose Up by observing that it performs two basic functions: A decision, where the packet's destination is computed and saved in s; and a conditional split, where the packet is relayed to one of the outputs or consumed based on s. We implement the first function using a decision block S (Fig. 5, *upper right*), which computes s. We implement the second function using an unconditional split, which copies a packet to both outputs paths, and two filters (F), which are configured to pass packets only for particular values of s.

Down reads the route encoded in an incoming packet's head (input on A) and decides whether to send it to its left daughter (output on L), its right daughter (output on R), or one of its two local memories (output on M_1 or M_2). It functions equivalently to Up—L maps to U and R maps to D—except in the case of a stop code. In that case, instead of consuming the packet, it delivers it either exclusively to a local memory (target mode) or to both daughters as well (flood mode). Its CHP reads:

$$\begin{array}{ll}
 & \mathsf{Down} \\
\equiv & h := \text{true}; \\
 & *[\ A?x; \\
 & \quad [\ h \rightarrow \\
 & \quad\quad dir := x.\text{route}[\text{msb}] \parallel \\
 & \quad\quad y.\text{route} := x.\text{route} \times 2 \parallel \\
 & \quad\quad y.\{\text{mode}, \text{mem}\} := x.\{\text{mode}, \text{mem}\}; \\
 & \quad\quad\quad [\ y.\text{route} = 0 \wedge y.\text{mode} = \text{t} \rightarrow \\
 & \quad\quad\quad\quad s := \text{target}; \\
 & \quad\quad\quad\quad [\ y.\text{mem} \rightarrow M_1!y \ [\!]\ \neg y.\text{mem} \rightarrow M_2!y\ ] \\
 & \quad\quad\quad [\!]\ y.\text{route} = 0 \wedge y.\text{mode} = \text{f} \rightarrow \\
 & \quad\quad\quad\quad s := \text{flood}; L!y \parallel R!y \parallel \\
 & \quad\quad\quad\quad [\ y.\text{mem} \rightarrow M_1!y \ [\!]\ \neg y.\text{mem} \rightarrow M_2!y\ ] \\
 & \quad\quad\quad [\!]\ y.\text{route} \neq 0 \wedge dir = \text{l} \rightarrow s := \text{left}; L!y \\
 & \quad\quad\quad [\!]\ y.\text{route} \neq 0 \wedge dir = \text{r} \rightarrow s := \text{right}; R!y \\
 & \quad\quad\quad] \\
 & \quad [\!]\ \neg h \rightarrow \\
 & \quad\quad [\ s = \text{target} \rightarrow \\
 & \quad\quad\quad [\ y.\text{mem} \rightarrow M_1!x \ [\!]\ \neg y.\text{mem} \rightarrow M_2!x\ ] \\
 & \quad\quad [\!]\ s = \text{flood} \rightarrow L!x, R!x, \\
 & \quad\quad\quad [\ y.\text{mem} \rightarrow M_1!x \ [\!]\ \neg y.\text{mem} \rightarrow M_2!x\ ] \\
 & \quad\quad [\!]\ s = \text{left} \rightarrow L!x \\
 & \quad\quad [\!]\ s = \text{right} \rightarrow R!x \\
 & \quad\quad] \\
 & \quad]; h := x.\text{tail} \\
 &]
\end{array}$$

As expected, this process is similar to Up, except for how a stop code (y.route $= 0$) is handled. In target mode (y.mode $=$ t), the packet is sent to either M_1 or M_2 (depending on y.mem). In flood mode (y.mode $=$ f), it is sent to L and R as well. It is important to note that once a packet begins branching it does so recursively at all daughter nodes—because the route fields of these branched packets are also stop codes ($y.\text{route} = x.\text{route} \times 2 = 0$).

[13]The stop condition should never hold true since routes are not supposed to terminate on the up path. However, since we can not guarantee this is the case, we explicitly check this condition.

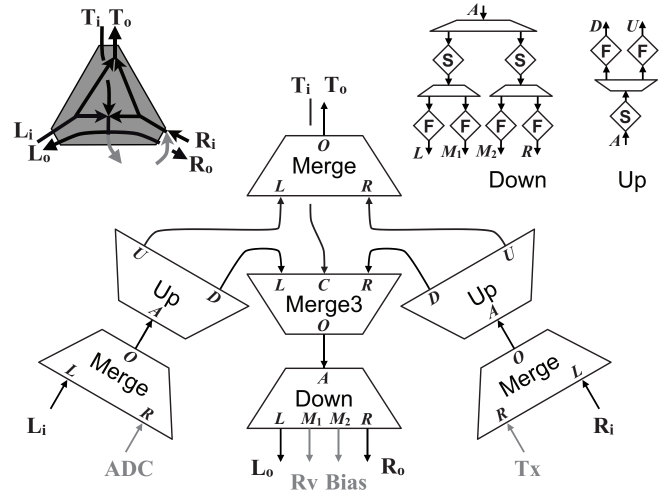

Fig. 5. Router Decomposition. The router (top left) communicates with two daughters (L_i, L_o, R_i, R_o), a parent (T_i, T_o), two local sources (grey) and two local sinks (grey). Its decomposition (center) combines upward and downward flowing traffic through merges (Merge, Merge3) that feed into conditional splits (Up, Down), which route or consume packets. Conditional splits are further decomposed (top right) into a decision (S), split, filter (F) cascade.

We decompose Down in a similar manner as Up, which results in a split–decision–split–filter cascade (Fig. 5, *upper right*). Alternative decompositions are possible, however our choice simplifies the design effort because Down is two instances of Up—albeit with slightly different control logic—with the addition of an extra split at the input.

B. Datapath implementation

To implement the router's four functional blocks (Decision, Split, Filter and Merge), we used a datapath template where control is relayed from bit to bit, flowing perpendicular to the data (Fig. 6) [33]. Bit-level transactions allow fine-grained pipelining in an asynchronous implementation, boosting throughput by eliminating completion trees (i.e., bits are not synchronized until their final destination). We paired bits together for energy efficiency, however, slicing the router's twelve-bit datapath into six bit-pairs. Each is communicated by a transition on one of four lines (1-in-4 code), half as many transitions as independent bits require (1-in-2 code). Furthermore, a single acknowledge line is used, instead of two [34].

Decision (S) is implemented with four logic blocks: HeadDetect, Shift, StopDetect, and Decide (Fig. 6a). HeadDetect determines when a headword is expected by monitoring the tail-bit ($h := x$.tail). Shift computes the new route ($y.\text{route} := x.\text{route} \times 2$) and extracts the direction ($dir := x.\text{route}[\text{msb}]$). StopDetect determines if the stop condition holds ($y.\text{route} = 0$). And Decide chooses one of three ($s := \text{stop}, \text{up}, \text{down}$) or four ($s := \text{target}, \text{flood}, \text{left}, \text{right}$) routing options, in the case of Up and Down, respectively. Instead of computing s explicitly, however, it decides what

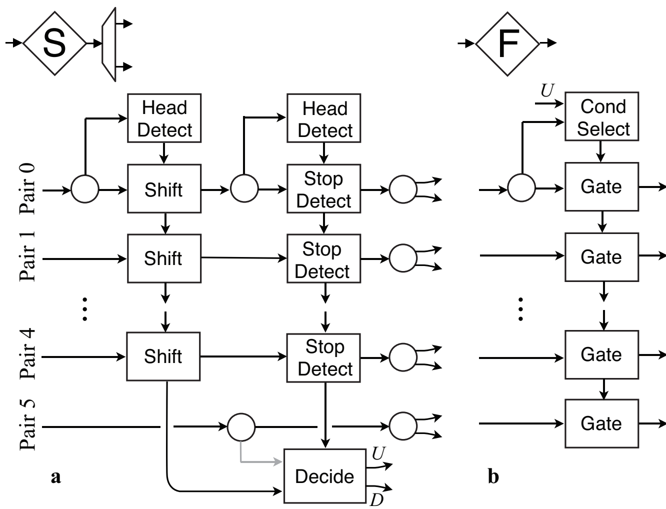

Fig. 6. Datapath for Up—Control Flows Perpendicular to Data. **a** Decision: HeadDetect monitors the tail bit and signals when a headword is received. Shift shifts the headword's bits (Pairs 0 through 4) and extracts the direction bit. StopDetect propagates a true signal if its bits are both zero, thereby determining if a stop code is present. Decide uses both pieces of information to determine the appropriate actions (filter or relay) two downstream filters (U and D) should perform. In Down, it interrogates the mode and memory bits as well (grey). Split (circles with two out-going arrows) copies the data to two paths. **b** Filter: CondSelect captures the desired action during the headword communication and Gate passes or filters bit-pairs accordingly.

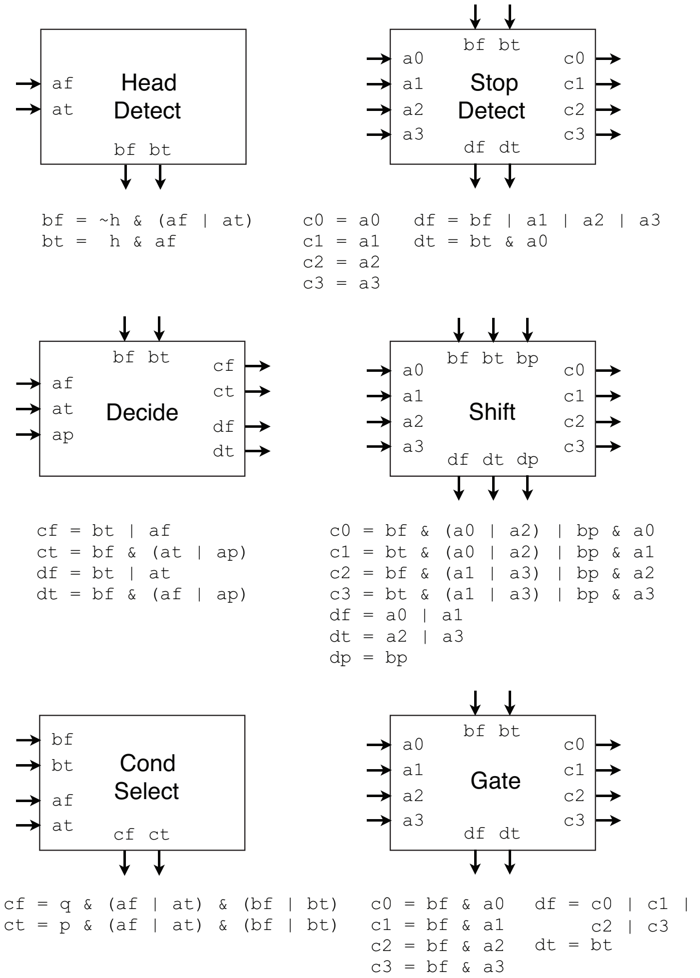

Fig. 7. Control and Data Logic Blocks (*left* and *right column*, respectively). HeadDetect: Sets state variable (`h`) when tail-bit is high (`at`), signals head (`bt`) for next word, resets state, and then signals payload (`bf`) for subsequent words. Decide: In Up, signals up-path to pass (`ct`) and down-path to filter (`df`) if direction bit is set (`at`), or up-path to filter (`cf`) and down-path to pass (`dt`) if it is reset (`af`). Unless a stop-code is present (`bt`), in which case signals both paths to filter. Communications (on `ct` and `dt`) triggered by payload-words (`ap`) are vacuous. CondSelect: Captures instruction during head communication (`bt` sets `q` or `bf` sets `p`), and instructs datapath to pass or filter accordingly (`cf` or `ct`, respectively). `q` and `p` are reset during tail communication (`at`). StopDetect: Passes data and, if both bits are zero (`a0`), propagates stop signal (`bt` sets `dt`). Shift: Shifts a 0 (`bf`) or 1 (`bt`) into outgoing bit-pair's LSB, shifts incoming bit-pair's LSB (= a1|a3) into outgoing bit-pair's MSB, and shifts incoming bit-pair's MSB (= a2|a3) out (`dt` or `df`). Or passes data unchanged (`bp`). Gate: Passes data (`bf`) or consumes it (`bt`).

action (filter or relay) the downstream Filters should perform. In Down's case, this decision requires interrogating the mode and memory bits.

Split is implemented with a single logic block (circles with one input and two outputs in Fig. 6a, *right*). It feeds packets to the two Filters, together with control signals from Decide that specify the appropriate action to perform.

Filter (F) is implemented with two logic blocks: CondSelect and Gate (Fig. 6b). CondSelect receives instructions from Decide during the headword communication. Gate performs the action instructed to every word in the packet, bit-pair by bit-pair, either passing the pair to the output or consuming it.

Merge is implemented with two logic blocks: SideSelect and Join (not shown). It is organized similarly to Filter, with CondSelect replaced by SideSelect and Gate replaced by Join [33]. Both blocks receive a bit-pair from each of two incoming paths—SideSelect arbitrates between them on a packet-by-packet basis and Join passes the chosen path's bit-pairs to its output.

While the twelve-bit datapath is sliced into six 1-in-4 groups (e.g., `a0 a1 a2 a3`), the perpendicular control path uses a 1-in-2 code (e.g., `bt bf`). These choices resulted in the logic-level implementations shown in Fig. 7 (see Table III in the Appendix for logic syntax definition). HeadDetect's `bt` and `bf` outputs are tied to StopDetect's `bt` and `bf` inputs, respectively. Thus, it propagates a stop signal when a headword is received and a go signal when a payload word is received. And HeadDetect's `bt` and `bf` outputs are tied to Shift's `bf` and `bp` inputs, respectively. Thus, it shifts in a 0 when a headword is received and does not shift when a payload word is received (Shift's `bt` input is not used). CondSelect's `ct` and `cf` outputs are tied to Gate's `bt` and `bf` inputs, respectively. Thus, it filters or relays all the packet's words. Merge is controlled similarly.

We implemented these logic blocks by writing down HSE for each one (i.e., choosing a specific signal transition sequence) and decomposing the HSE into a PRS (i.e., a set of boolean expressions that enforce the specified order of signal transitions). Each production rule corresponds to a gate's pull-up or pull-down (depending on the signal transition's sign). The HSE, logic simulation and circuits for two of these blocks (HeadDetect and StopDetect) is provided in the Appendix for interested readers (Fig. 15). Due to space limitations, we are

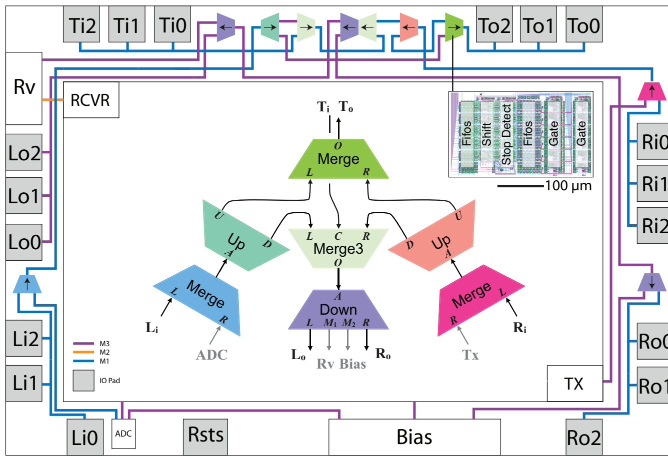

Fig. 8. Floorplan with Sample Datapath Layout. The router's logic elements are placed within a thin ring around the chip's periphery and connected as shown by the color-coded schematic diagram (middle). Each of its three bi-directional ports (Li Lo, Ti To and Ri Ro) uses three pad-groups. Each pad-group has seven IO-pads (2 power, 4 signal, 1 enable). The layout for the datapath that implements Decision and Split (Fig. 6a) and two Filter paths (Fig. 6b) is shown (inset).

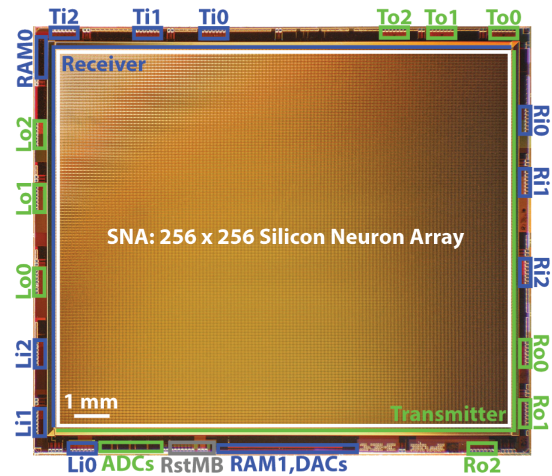

Fig. 9. Neurocore. A 12 × 14 sq-mm die fabricated in a 180nm CMOS process integrates: SNA, a 256×256 silicon-neuron array. RAM0, a 256×16-bit SRAM that specifies target synapse type or no connection. RAM1, an SRAM that specifies eighteen configuration bits and sixty-one analog biases—common to all the Neurocore's silicon neurons. DACs, sixty-one digital-to-analog converters that produce the analog biases. RstMB, five reset signals and a master bias circuit that generates the DACs' reference current. ADCs, four analog-to-digital converters that digitize any selected neuron's internal analog signals. Li Lo, Ti To and Ri Ro, input or output ports—organized in three groups of seven pads—that support bidirectional communication with the Neurocore's parent and two daughters. The unlabeled pads power digital or analog circuitry.

unable to provide transistor-level schematics for all the blocks that make up the router. However, we do provide PRS for the six blocks shown in Fig. 7 in the Appendix (Fig. 16).

IV. Physical Design

The physical design followed a full-custom flow, where each block was layed-out by hand, since automated place-and-route tools were not readily available for our asynchronous design style. Although hand layout was tedious, it allowed for an efficient bit-sliced design that closely followed the datapath description (see layout inset in Fig. 8).

A. Router floorplan

The main challenge in floor-planning the router was mapping its tree-like structure into the thinnest possible ring around the chip's perimeter (called the crust), and wiring all the components together (Fig. 8). To minimize the router's area, all datapath blocks were placed within the IO-pad ring, utilizing a region that is traditionally unused. Router blocks were placed to minimize connection lengths between blocks. For instance, the merge that combines L_i- and ADC-traffic sits on the lower left while the one that combines R_i- and transmitter-traffic sits on the upper right. FIFOs were interspersed between router blocks to provide queueing and to boost throughput by breaking up long buses; their number was determined by the area available. Placement and wiring was particularly challenging at the top of the chip where all router paths converge, causing the greatest congestion.

The entire router was wired using only four routing channels (two on layer M1 and two on layer M3), each containing six 1-in-4 groups that lie on the crust's inner periphery. Blocks were placed and wired using custom placement and wiring scripts. Several wiring channels between segments extend long distances (>300μm), and we employed two techniques to minimize the effect of crosstalk and ensure good signal integrity. First, wiring channels were organized so 1-in-4 groups were shielded from each other (as well as the acknowledge), limiting crosstalk to within a group where only one signal transitions at a time. Second, cross-coupled active pulldowns were placed within each group—pulling down a group's other three lines when one was raised.

B. Chip I/O

An additional challenge was to accommodate the router's three bidirectional ports in a standard 180-ball wire-bonded package. Without any serialization, each bidirectional port requires 84 pads (48 data + 12 enables + 24 power, implementing twelve 1-in-4 groups), which total 252 pads for all three ports. Therefore, to fit in the desired package, we implemented two-to-one multiplexing across 1-in-4 groups, reducing the number of pads to 42/port and cutting the total IO-pad count to 126. To maintain the same bandwidth with half the pads, we send data on every transition with a two-phase 1-change-4 (1c4) protocol (see [35] for specifics), which is twice as efficient as the four-phase 1-in-4 protocol used within the chip. Our custom input and output pads included circuitry to decode and encode 1c4, respectively. We placed multiplexor and demultiplexor blocks adjacent to the pads. Each port's pads were largely placed on its corresponding side of the chip (left, top, or right) to facilitate building a multichip PCB with straight-shot connections between adjacent chips.

C. Fabrication

The router was part of the Neurocore chip, which was fabricated in IBM's 180nm mixed-signal process (Fig. 9). The

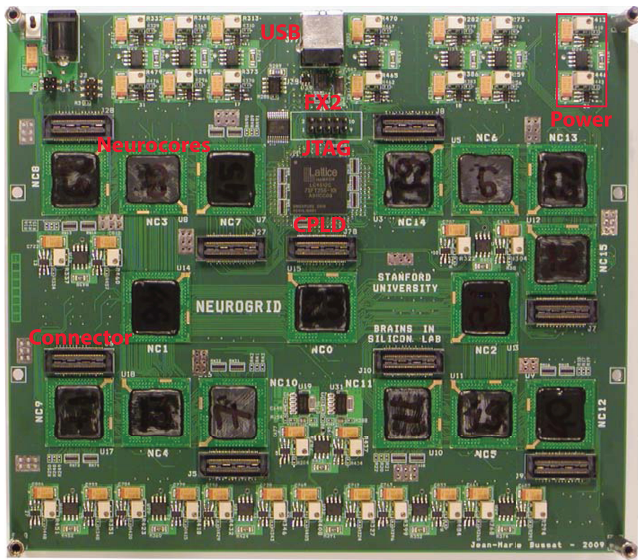

Fig. 10. Neurogrid. A 6.5×7.5 sq-in circuit board assembles sixteen packaged Neurocore dies into a binary tree with 91M spikes/s links (peak rate). This system models spiking neural networks with up to a million neurons, organized in up to 16 layers, each with up to 256×256 neurons. Neurocores are rotated to allow straight shot connections as shown in Fig. 1b. The board also has a CPLD (*Lattice* ipsMACH LC4512C), a USB interface chip (*Cypress Semiconductor* FX2), regulated digital and analog power supplies for each Neurocore (Power), parallel IO connectors (Connector) at the tree network's leaves and root, and USB and JTAG connectors (see Fig. 11 for a simplified schematic). The back of the board holds tie-down resistors, bypass capacitors and jumpers; it has no active components.

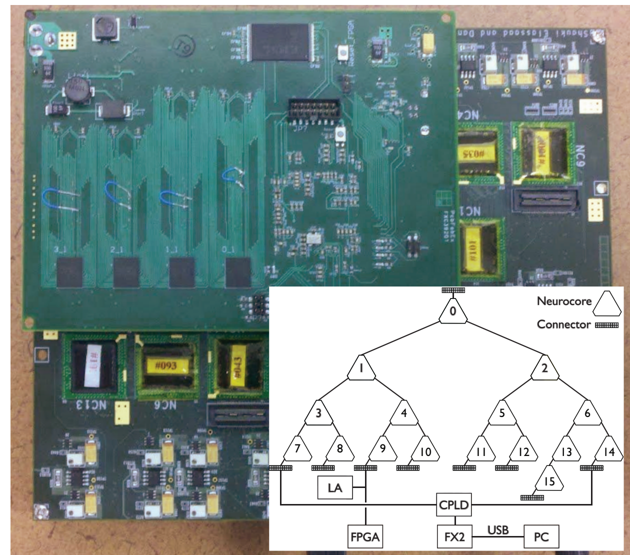

Fig. 11. Daughterboard mounted atop Neurogrid's main board. Its SRAMs (*Cypress Semiconductor* CY7C1071DV33) store 32 MBytes with a 50 ns access time for 4 bytes, which specify a target neuron's chip, row, and column addresses. A FPGA (*Xilinx* Spartan-3E, 100MHz clock) calculates a base address from which to begin reading SRAM entries from the source neuron's chip, row, and column address, parsing the incoming packet to obtain this information. Four SRAM chips are visible (lower left); there are four more on the other side, together with the FPGA. To characterize the router's performance, probe packets injected by the PC were relayed through the FX2 and the CPLD (*insert*), routed up from Neurocore 7 to Neurocore 0, then down to Neurocore 9, and then on to the daughterboard (moved from the root). It's FPGA acknowledged Neurocore 9; the logic analyzer's (LA) probe recorded these communications.

12×14 mm^2 die was dominated by the 256×256-silicon-neuron core (22 million transistors), with the router and its 180 pads confined to the periphery (over 1 million transistors). The pads were wire-bonded to a custom two-layer substrate with short traces to keep inductance below 4 nH (keeping LdI/dt perturbations within acceptable margins).

Sixteen Neurocores were assembled on a 6.5×7.5 sq-in printed circuit board to build Neurogrid, a neuromorphic system designed to simulate a million cortical neurons connected by billions of synapses in real-time (Fig. 10). To connect each of its million neurons to about five thousand others, Neurogrid uses multicast routing together with unicast routing and analog signaling. Unicast routing realizes arbitrary connectivity by translating each incoming packet to about eight outgoing packets using a FPGA daughterboard with 32MB of SRAM. This daughterboard mounts to Neurogrid via the connector that sits above the root chip (Fig. 11). Multicast routing realizes translation-invariant connectivity between corresponding locations on about six Neurocores using the 256×16-bit SRAM in each Neurocore. And analog signaling realizes connections to about a hundred neighboring neurons on the same Neurocore using a diffusor network with programmable space-constant [13]. These three routing mechanisms realize the cortex's columnar and topographic organization efficiently [36].

V. Test Results

We tested the fabricated chips to verify the router's functionality, and found the design to be fully functional with good yield. Individual Neurocores were first verified using a single-chip testboard with a zero-insertion-force socket. Packets were injected into the router from a PC to exercise all the internal datapaths. Then router outputs were looped back to inputs (via programmable logic on the PCB) to emulate multichip network traffic and packet contentions. Of 216 packaged die tested, 167 passed these router tests, a 77.3% yield. Further tests were performed on these 167 die to verify that their analog circuits and analog–digital interfaces were functional; the ultimate test configured the chip's neurons to synchronize through recurrent inhibition. Seventeen (17) die failed these tests (e.g., overheated or did not produce spikes) and 23 were only partially functional (e.g., displayed gradients across the array or defects in certain portions), leaving 127 fully functional die, an overall yield of 59%. Neurocores with full functionality (including a working ADC, DAC, etc.) were selected for populating Neurogrid boards. In total, five Neurogrid boards were assembled; all experiments described in this paper were performed on one of the boards (shown in Fig. 10).

A. System bring-up and functional testing

We interact with Neurogrid through a custom graphical user interface (GUI) that allows us to configure its Neurocores individually, visualize their activity, and stimulate them. The board includes a *Cypress* FX2, which supports USB 2.0 communication with a PC, and a *Lattice* CPLD, which interfaces between the Neurocores' 1c4 ports and the FX2's 16-bit IO port (see Fig. 11). System bring up entails:

1) Reset the entire system (toggle resets via CPLD, software controlled).
2) Initialize each Neurocore's synaptic connectivity (SRAM), neuron and synapse parameters (DACs), and desired routes for spike packets to their default states (e.g., no spiking). Programming packets are sent to each chip through the router network sequentially.
3) For jitter tests, configure a daughterboard's FPGA as a packet sink and attach a logic analyzer connector to sniff packets.

Once the system is initialized, the analog parameters of each Neurocore are configured through the GUI so neurons generate spontaneous spiking activity, injecting spike packets into the network with their programmed route.

We found that a Neurocore could output spikes at a maximum rate of 43.4M spikes/s—or 663 spikes/s per neuron, on average. This rate is dictated by its neural array's transmitter circuitry (see [7, 37] for detailed descriptions of the transmitter). The transmitter takes 86 ns to transfer a row's spikes in parallel to the neural array's periphery, and encodes the row's address (12-bit word) at the same time. Then, it takes 23 ns to encode a column address for each spike read from that row. While this column address encoding is occurring, the next row's spikes are read out of the array. Hence, the 86 ns it takes to read them is hidden if the previous row had more than three spikes. Thus, a maximum transmission rate of 43.4M spikes/s (the rate at which column addresses are encoded) is achievable. These row and column cycle-times were determined by finding the shortest interval between short packets (< 3 column addresses) and the average interval between latter column addresses in long packets (> 3 column addresses), respectively.[14] The transmitter's peak output rate was less than a link's (see below), so we had to merge traffic from several chips in order to measure a link's performance.

To test the router's multicast capability, and thereby demonstrate our central claim of deadlock-free, multicast routing, we configured Neurogrid to simulate a neural network with fifteen cell-layers. The layers were arranged in a ring—the first and last were neighbors—each layer connected to its three nearest-neighbors on either side, as well as to itself (Fig. 12). This connectivity pattern was implemented using the router's multicast function. With the spatial decay constant we programmed for the analog diffusors, each neuron received inhibition decaying exponentially with distance such that 8000 neurons (in a cylinder seven-layers thick with a nineteen-neuron radius) contributed 50% of its total inhibition. Such recurrent connectivity patterns are expected to give rise to globally synchronous spike activity, which is what we observed, confirming that the connectivity was implemented correctly by the router's multicast capability.[15]

[14] The first two column addresses are output at a faster rate because they are encoded at the same time the route, chip address and row address are being output, thereby hiding the column's cycle-time.

[15] The average spike rate was kept low (0.42 spikes/s) to allow spikes from all the neurons to be logged over USB, which can carry up to a million spikes/s (limited by our current CPLD firmware).

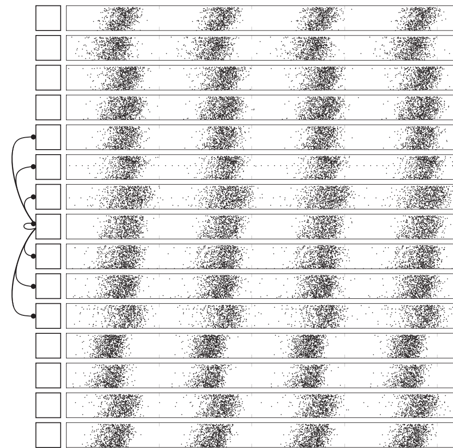

Fig. 12. Multicast Capability Demonstration. Simulation results of a neural network with fifteen 256×256 cell layers—983,040 neurons in total. Spike rasters (from a tenth of each layer's neurons) reveal global synchrony, as expected from the network's recurrent inhibition. Each cell layer's neurons inhibit themselves as well as neurons in three neighboring layers to either side (the central layer's connectivity is shown). The layers are arranged in a ring, so the first and last layers are nearest neighbors. The synchronized activity was rhythmic, with a frequency of 3.7 Hz; the neurons fired 0.42 spikes/s on average. The tick marks are 250ms apart.

B. Router performance

The router's performance is characterized by the mean and standard-deviation of its latency distribution. However, except for the heaviest loads, the mean latency was largely independent of the load. The reason being that the propagation delay of the 100 to 300 FIFOs traversed in each chip (each FIFO adds approximately 1.2ns) dominates the mean latency. For instance, the latency through a single chip—from port Li, through its up path's merge and split, through its down path's merges and splits, to port Lo—was 181ns.[16] According to the theory (see below), the latency due to queuing equals this FIFO latency when the load is 93%. Therefore, we focused on measuring the latency distribution's standard-deviation. Specifically, we measured the standard deviation of the intervals between packets injected into the network at equal intervals, defined as the jitter. To measure the worst-case jitter, we programmed the computer to send these packets—called *probe packets* to distinguish them from traffic packets—over the longest route in Neurogrid's tree network at 135μs intervals (see Fig. 11). The FPGA daughterboard acknowledged receipt of these probe packets while a logic analyzer recorded their time of arrival with 125ps resolution (*Tektronix* TLA7012).

To measure the jitter's dependence on the load, the eight Neurocores at the tree's leaves (ignoring Neurocore 15) were programmed to generate packets at a rate of λ_{chip} words/s and route them up to Neurocore 0 (the root) and then down to Neurocore 9. This traffic pattern loaded the link connecting

[16] These measurements and those of the interNeurocore link's speed, mentioned at the end of this section, were made on a board with half of its Neurocores replaced by breakouts to build an eight-chip network. The breakouts connected neighboring Neurocores together while providing access to the signals running between them (at test points).

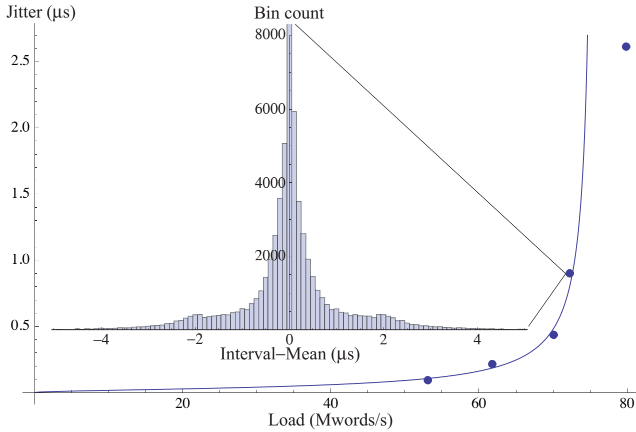

Fig. 13. Jitter measurements and fit with theory. The jitter (t_{jitter}) is plotted for various loads (λ_{total}) and fit with the theoretical prediction (dots and line, respectively; $\mu_{\text{link}} = 75.7$M word/s). For instance, for a load of 72.2M word/s, the probe-packet interval distribution (inset) has 0.9μs standard deviation, defined as the jitter. The bins are 100ns wide. The central one had 53,480 of a total of 104,855 intervals; it was cut off at 8000 to show more detail. Note that the last jitter datapoint deviates from the theory due to load-shedding, which occurs because of queueing in the neural array. That is, as a neuron spends more time waiting for its spikes to be read, its interspike interval lengthens by that amount of time, resulting in a lower-than-expected spike rate, or load.

level i to level j with

$$\lambda_{i,j} = \begin{cases} 2^i \lambda_{\text{chip}} & j > i \text{ (upward link)} \\ 2^3 \lambda_{\text{chip}} & j < i \text{ (downward link)} \end{cases} \tag{1}$$

words/s, where levels are numbered from 0 (leaves) to 3 (root). Thus, the probe packet experienced a traffic pattern identical to that produced if the traffic packets were multicast to all sixteen chips by flooding at the root. However, unlike with flooding, the relatively slow FPGA that acknowledged communications on Neurocore 9's left port (at 20ns/word) was not overwhelmed because traffic packets were not ejected.

The jitter can be predicted by calculating the variance each link adds and summing these terms over all the links the probe packets traversed. Assuming arrival and service times are Poisson-distributed, the mean and variance of the number of cycles spent waiting at each link is given by [38]

$$\langle w(i,j) \rangle = \frac{\lambda_{i,j}/\mu_{\text{link}}}{1 - \lambda_{i,j}/\mu_{\text{link}}}$$

$$\langle (w(i,j) - \langle w(i,j) \rangle)^2 \rangle = \frac{\lambda_{i,j}/\mu_{\text{link}}}{(1 - \lambda_{i,j}/\mu_{\text{link}})^2}$$

where μ_{link} is the link's capacity (i.e., its mean cycle-time is $1/\mu_{\text{link}}$), which, in contrast to the varying loads, is the same for all links. Taking the square-root of twice the total variance yields the jitter:

$$t_{\text{jitter}} = \frac{\sqrt{2}}{\mu_{\text{link}}} \sqrt{\frac{4p}{(1-p)^2} + \frac{p/2}{(1-p/2)^2} + \frac{p/4}{(1-p/4)^2}} \tag{2}$$

where $p = \lambda_{\text{total}}/\mu_{\text{link}}$ expresses the total traffic, $\lambda_{\text{total}} = 8\lambda_{\text{chip}}$, as a fraction of the link capacity. As expected, the jitter diverges when the total traffic exceeds the link capacity (i.e., $p > 1$).

Note that while the jitter is measured for (fully transmitted) probe packets, the traffic is measured in word/s, not packet/s. Traffic includes a mixture of non-burst-mode packets (i.e., single-spike packets, which have five words) and burst-mode packets (i.e., multiple-spike packets, which have additional column address words). It is not possible to limit the measurements to the former because burst-mode packets occur probabilistically. The probability increases with the load each chip generates, and thus depends on how the total load is distributed across chips. For instance, a higher burst rate will occur if the same total load was distributed among a smaller number of chips. Thus, the resulting word-rate will be lower, and the jitter (or latency) will be lower. Characterizing jitter in terms of word-rate avoids this ambiguity, because the interchip link's performance is specified in word/s.

Our experimental measurements of t_{jitter}'s dependence on λ_{total} agreed well with our theoretical predictions (Fig. 13). We obtained a good fit with μ_{link}, the one free parameter, set to 75.7M words/s, suggesting that a link could communicate no more than one word every 13.2ns. This cycle-time appears to be limited by the pads, not the datapath. The datapath's cycle-time is no more than 6.6ns/word while the pad-to-pad cycle-time is 11ns/word. The former was determined by measuring the latency distribution of probe packets that sometimes collided with transmitter packets at a merge; the latter was determined from peaks in the inter-word interval distribution captured with a logic analyzer probe between two chips at the tree's lowest levels. Thus, our fit's 13.2ns/word link capacity is a little slower than the pads but much slower than the datapath. The difference between the fitted 13.2ns/word link capacity and the measured 11ns/word pad-to-pad cycle-time is probably due to the difference in the length of PCB traces connecting chips near the tree's leaves (0.6cm) and near its root (3.3cm).

Given the specification that no packet should experience more than 1μs of jitter, the tree network can deliver over a billion words/s. We came to this conclusion by using our theoretical fit to interpolate the load at which a packet traveling the longest route in the network experiences 1μs jitter. This procedure yielded $\lambda_{\text{total}} = 73.0$M words/s, a load that corresponds to 96% of $\mu_{\text{link}} = 75.7$M words/s, the fitted link capacity. Multiplying further by 16 to account for these words being multicast from the root to all 16 chips yields a delivery rate of 1.17G words/s. Note that a spike is represented by a single word at high traffic levels—where the transmitter appends additional column addresses to a packet for each additional spike read from the same row (called burst-mode).

C. *Router power*

Neurogrid's mainboard drew 0.876A at 3V (2.63W) with one million neurons spiking at 0.7Hz. This activity was measured by unicast routing every spike to the PC, each one traversing 4 links on average.[17] The current dropped by 12mA when we reduced the spike rate to 0.06Hz, indicating that the digital communication consumed negligible power. Dividing

[17]The CPLD–FX2 interface can only handle up to 1 M spikes/s in its current implementation, which limited the average spike rate to 0.7Hz, as headroom must be left to accommodate rate fluctuations.

the change in power consumption (36mW) by the change in total spike rate (0.64 MHz) as well as the number links traversed per spike (4) yields an energy efficiency of 14nJ per spike per link. Note that changing the on-chip tunable diffusor networks does not impact power consumption, since the diffusor simply redistributes the current to implement fanout over a larger area [13].

The daughterboard consumed 0.32W on standby, and this increased to 0.49W to perform 64 memory lookups for each incoming spike. Each lookup provides a target's address as well as a weight that represents the probability that the spike is sent to this target [39]. In our experiment, all the weights were set to a probability of 1/16, which resulted in 4 outgoing spikes (to the mainboard) per incoming spike (to the daughterboard), on average. Dividing the total power consumption (0.32W+0.49W) by the frequency of memory look-ups (64×1MHz) yields an energy efficiency of 12.6nJ per memory lookup.[18]

In addition to characterizing the efficiency of implementing fanout through multicast communication on Neurogrid's mainboard or address translation on Neurogrid's daughterboard, these energy measures enable us to quantify the energy savings for different combinations of mainboard-level multicast fanout (14nJ per spike per link) and daughterboard-level unicast fanout (12.6nJ per memory lookup). For each memory look-up avoided by using multicast fanout, 12.6nJ is saved plus 14nJ times the path length (in links) from the root to the lowest common ancestor.

VI. Discussion

The design that we have presented represents a major advance for building multicast and deadlock-free routers. Our compact design is small enough to fit in the periphery of each chip, allowing chips to connect together seamlessly to construct multichip networks. As a concrete example, we used our router as the communication backbone for Neurogrid, a fully functioning sixteen-chip neuromorphic system that is the first with one-million spiking silicon neurons operating in real time. This system consumes 3.5W (mainboard plus daughterboard) thanks to an asynchronous implementation. To match the interchip link's cycle time (6.6ns per half word), a synchronous design would require a 150MHz clock. We estimate that distributing this clock across 16 chips would consume 10W in active power.

Our multicast approach is more scalable than previously proposed mesh networks that rely on unicast routing where fanout is implemented by sending a separate packet to each target ($O(n)$ scaling verses $O(n^{3/2})$ for all-to-all traffic). In particular, multicast enabled our router to deliver 1.17G words/s across sixteen chips with 1μs jitter—12.9 times a link's peak bandwidth (of 91M words/s) and 4.3 times a chip's total input-to-output bandwidth (of 273M words/s)—effectively overcoming the bottleneck at the tree's root. Furthermore, unlike a bus or 1D network [10, 11], our tree router can take advantage of spatial clustering by mapping clusters onto subtrees. Such clustering is observed in the cortex where axons branch close to their targets (c.f., multicast), as opposed to branching near their soma (c.f., unicast).

To the best of our knowledge, our router is the first to ensure deadlock-free multicast communication (i.e., no router configuration can result in deadlock). Existing router designs for large-scale multichip neuromorphic systems fall into two categories: flat or hierarchical topologies.[19] SpiNNaker [41] (a mesh network of degree 6)[20] uses a flat topology that can lead to deadlock while multicasting.[21] While deadlock-free multicast may be achieved by implementing a virtual tree on a mesh, this software-based approach results in a scaling that matches multicast tree networks (see Table 1) while using more resources (nodes of degree 4 versus degree 3). In addition, implementing the virtual tree requires memory lookups enroute, resulting in longer latencies. FACETS [42] (a two-level network with degrees 9 and 6) and HiAER [43] (a two-level network with degrees 2 and 5) use a hierarchal topology but are not guaranteed to be deadlock-free because branching is allowed on the upward path. Hardware measurements of throughput versus latency were not available from any of these systems for comparison at the time of publication.

Moving beyond neuromorphic systems, we believe our router can also be used in other domains. For example, our router has a flexible communication protocol that allows arbitrary length payload, which can support any data format. Also, the sender can inject a packet into the network without knowing its length, which is useful for streaming data in real time. Also networks that require more arbitrary connections than the daughterboard approach provides may be supported by expanding the on-chip SRAMs' capacity. These SRAM's translate packets in parallel, providing scalable bandwidth. This flexibility and scalability, in addition to its low-latency multicasts and low-power consumption, make our design an attractive candidate for a wide range of multichip systems.

VII. Appendix

A. Packet Types

The packet types the router handles include Spike, Connect, Bias, and Sample (Fig. 14). These types are processed at their targets based on two operation bits: Memory (M) and Write (W): M determines whether the packet is sent to the SRAM that specifies synaptic connectivity (Spike or Connect) or to the SRAM that specifies neuronal parameters (Bias). W determines whether the packet's data (third word) is written to the specified address (second word). Sample packets contain digitized samples of particular analog signals recorded from

[18]Dividing by 64 instead of 4 is justified because the energy use is dominated by transferring addresses between the FPGA and the SRAM chips on the daughterboard.

[19]We do not compare with on-chip networks such as ConvNet [40], a system with two FPGAs, each implementing a mesh network of degree 4.

[20]Degree is defined as the number of bidirectional links each node has.

[21]SpiNNaker detects deadlock by monitoring the amount of time packets spend queueing; packets that wait longer than a set maximum time are dropped. To minimize the number of packets dropped, their network is designed for the worst-case traffic pattern and is then underutilized during normal operation. In our view, this design choice will lead to higher power consumption due to larger devices and higher leakage when compared with our deadlock-free approach.

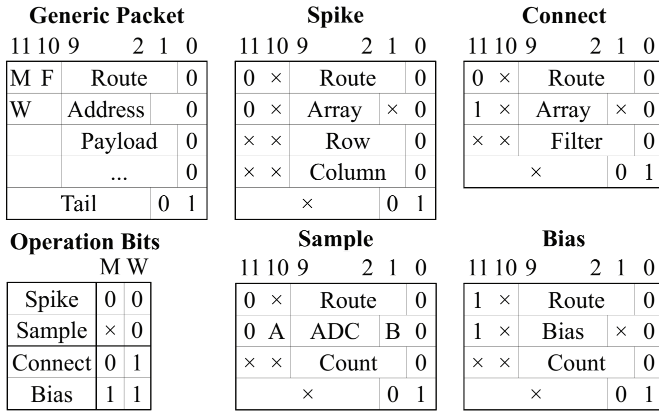

Fig. 14. Packet Format and Various Types. **Generic packet**: Consists of a route, an address, an arbitrary number of additional words, and a tail. **Operation bits**: Specify whether to deliver the packet to the SRAM that stores synaptic connections (M = 0) or the one that stores neuronal parameters (M = 1) and whether to perform a read (W = 0) or a write (W = 1). **Spike**: Used to communicate (W = 0) a spike from a source array (Array) to a target array (Route)—exclusively (F = 0) or to all its descendants as well (F = 1). May have multiple column addresses (burst-mode) **Connect**: Used to program (W = 1) the synaptic connectivity SRAM (M = 0) of a target array (Route) to filter or deliver (Filter) spikes from a specified source array (Array). **Sample**: Used to communicate (W = 0) a digitized sample (Count) of a preselected analog neuronal signal. Two additional bits (A and B) specify which of four ADCs on that chip the sample came from—they share the same base address (ADC). **Bias**: Programs (W = 1) target's neuronal parameter SRAM (M = 1) to set the specified bias (Bias) to the value given (Count).

TABLE III
HSE AND PRS PRIMITIVES

Operation	Notation	Explanation
Signal	`v`	Voltage on a node
Complement	`~v`	Inversion of `v`
And	`v & w`	`&` takes precedence
Or	`v | w`	over `|`
Set	`v+`	Drive `v` high
Clear	`v-`	Drive `v` low
Wait	`[v]`	Wait till `v` is high
Sequential	`u -> v+`	`[u];v+` in HSE
Concurrent	`v+,w+`	`v+,w+` in HSE
Repetition	`*[...]`	Just like in CHP

a preselected neuron, such as its membrane potential or the postsynaptic currents of one of its four synapse types.

B. Example Circuits and Production Rules

We provide schematics, together with logic simulation and HSE for a control and a data block (Fig. 15). In addition, we also provide production rules for the six blocks shown in Fig. 7 (Fig. 16). Please refer to Table III for a description of HSE and PRS primitives.

ACKNOWLEDGMENT

The authors thank Samir Menon for helpful discussions on the tree architecture; Anand Chandrasekaran for help with pad and ADC design; Daniel Neil for help with PCB design; Swadesh Choudhary for help with the daughterboard design; and Ben Benjamin for help with the CPLD as well as data collection, analysis, and rendering (for Fig. 12). We also thank Kimberly Chan for administrative support. In addition, we thank Tobi Delbrück and Rajit Manohar for providing layouts for the bias generator and SRAM, respectively.

REFERENCES

[1] C Mead. *Analog VLSI and Neural Systems*. Addison Wesley, Reading MA, 1989.
[2] H Esmaeilzadeh, E Blem, R St. Amant, K Sankaralingam, and D Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.
[3] A. Cassidy and A.G. Andreou. Dynamical digital silicon neurons. In *Biomedical Circuits and Systems Conference, 2008. BioCAS 2008. IEEE*, pages 289–292, 2008.
[4] P Merolla, J Arthur, F Akopyan, N Imam, R Manohar, and D Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE, 2011.
[5] M Mahowald. *An Analog VLSI System for Stereoscopic Vision*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.
[6] M Sivilotti. *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. Phd thesis, California Institute of Technology, 1991.
[7] K A Boahen. A burst-mode word-serial address-event link I: Transmitter design. *IEEE Trans. Circ. & Sys. I*, 51(7):1269–1280, 2004.
[8] K A Boahen. A burst-mode word-serial address-event link II: Receiver design. *IEEE Trans. Circ. & Sys. II*, 51(7):1281–1292, 2004.
[9] J Lin, P Merolla, J Arthur, and K Boahen. Programmable connections in neuromorphic grids. In *in 49th IEEE Midwest Symposium on Circuits and Systems*. IEEE Press, 2006.
[10] S Bamford, A Murray, and D Willshaw. Large developing receptive fields using a distributed and locally reprogrammable address-event receiver. *Neural Networks, IEEE Transactions on*, 21(2):286–304, 2010.
[11] Paul A Merolla, John V Arthur, Bertram E Shi, and Kwabena A Boahen. Expandable networks for neuromorphic chips. *IEEE Transactions on Circuits and Systems I Regular Papers*, 54(2):301–311, 2007.
[12] G N Patel, M S Reid, D E Schimmel, and S P DeWeerth. An asynchronous architecture for modeling intersegmental neural communication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(2):97–110, 2006.
[13] A Andreou and K Boahen. Translinear circuits in subthreshold mos. *Analog Integrated Circuits and Signal Processing*, 9(2):141–166, 1996.
[14] K Zaghloul and K Boahen. Optic nerve signals in a neuromorphic chip i: Outer and inner retina models. *Biomedical Engineering, IEEE Transactions on*, 51(4):657–666, 2004.
[15] P Lichtsteiner, C Posch, and T Delbruck. A 128x128 120 db 15 us latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid State Circuits*, 43(2):566–576, 2008.
[16] B Wen and K Boahen. A silicon cochlea with active coupling. *Biomedical Circuits and Systems, IEEE Transactions on*, 3(6):444 –455, dec. 2009.
[17] T Choi, P Merolla, J Arthur, K Boahen, and B Shi. Neuromorphic implementation of orientation hypercolumns. *IEEE Transactions on circuits and Systems-I: Regular Papers*, vol. 52(3):pp. 1049–1060, June 2005.
[18] Rafael Serrano-Gotarredona, Teresa Serrano-Gotarredona, Antonio Acosta-Jiménez, Clara Serrano-Gotarredona, José A Pérez-Carrasco, Bernabé Linares-Barranco, Alejandro Linares-Barranco, Gabriel Jiménez-Moreno, and Antón Civit-Ballcels. On real-time aer 2-d convolutions hardware for neuromorphic spike-based cortical processing. *Neural Networks, IEEE Transactions on*, 19(7):1196–1219, 2008.
[19] J Arthur and K Boahen. Recurrently connected silicon neurons with active dendrites for one-shot learning. In *IJCNN'04 International joint conference on neural networks*, pages 1699–1704. IEEE Press, 2004.
[20] R Serrano-Gotarredona, M Oster, P Lichtsteiner, A Linares-Barranco, R Paz-Vicente, F Gomez-Rodriguez, L Camunas-Mesa, R Berner, M Rivas-Perez, T Delbruck, Shih-Chii Liu, R Douglas, P Hafliger, G Jimenez-Moreno, A Ballcels, T Serrano-Gotarredona, A Acosta-Jimenez, and B Linares-Barranco. Caviar: A 45k neuron, 5m synapse, 12g connects/s aer hardware sensory 2013 processing learning actuating system for high-speed visual object recognition and tracking. *Neural Networks, IEEE Transactions on*, 20(9):1417 –1438, sept. 2009.
[21] J Navaridas, M Luján, J Miguel-Alonso, L Plana, and S Furber. Understanding the interconnection network of spinnaker. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 286–295, New York, NY, USA, 2009. ACM.

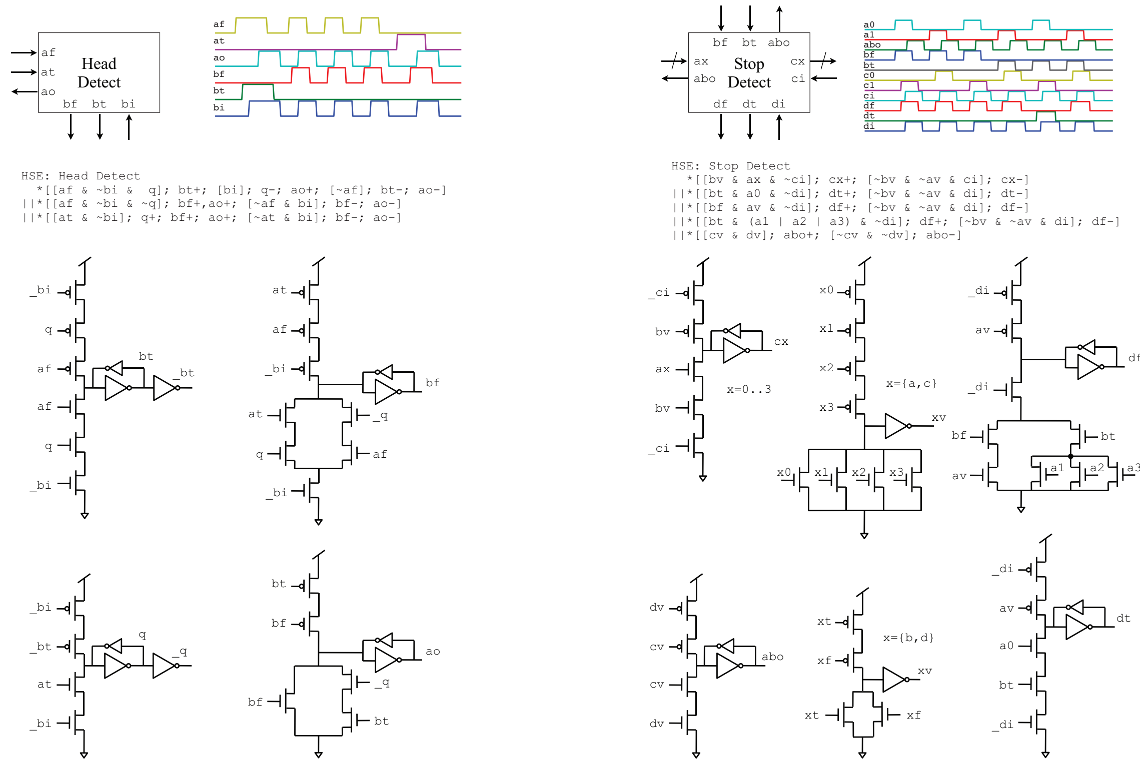

Fig. 15. Implementation of HeadDetect and StopDetect. Logic simulation (*top*), HSE (*middle*), and circuits (*bottom*) illustrate the synthesis procedure. Please refer to Fig. 7 for the functions of these blocks. The synthesis shown here is one of many reshufflings (i.e., reorderings of signal transitions); the particular reshuffling that we used is specified by the HSE and evident in the logic simulation. Based on these reshufflings, a particular choice of signal polarities (active low or active high) resulted in the transistor-level schematics shown. We use "_" to indicate an active-low signal (e.g., `_bi`). Note that weak feedback inverters (staticizers) are used to hold state where necessary.

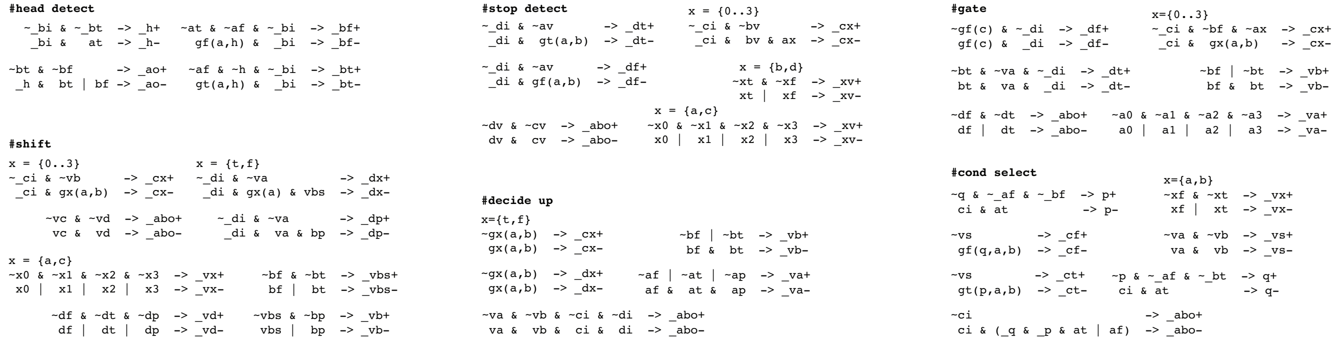

Fig. 16. Production Rule Sets (PRS) for the Six Router Blocks. The PRS's shown here, which were obtained from the HSE reshufflings chosen and the logic functions defined in Fig. 7, were translated into transistor-level schematics (shown for HeadDetect and StopDetect in Fig. 15). The logic function `gx(a,b)` in the production rule `... & gx(a,b) & ... -> _cx-` is specified in Fig. 7 as `cx=gx(a,b)`, except in the case of StopDetect, where `gf(a,b) = bf & av | bt & (a1 | a2 | a3)`. For example, in HeadDetect's pull-down for `_bt`, the logic function `gt(a,h) = h & af`.

[22] C Zamarreno-Ramos, A Linares-Barranco, T Serrano-Gotarredona, and B Linares-Barranco. Multicasting mesh aer: A scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. *Biomedical Circuits and Systems, IEEE Transactions on*, PP(99):1, 2012.

[23] W Dally and B Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco CA, 2004.

[24] R Boppana, S Chalasani, and C Raghavendra. On multicast wormhole routing in multicomputer networks. In *In Symposium on Parallel and Distributed Processing*, pages 722–729, 1994.

[25] A Despain and D Patterson. X-tree: A structured multiprocessor computer architecture. In *Proc. IEEE 5th Annu. Symp. Comput. Arch.*, pages 144–51, 1978.

[26] S Browning. The tree machine: A highly concurrent computing environment. Technical Report Caltech-CS-TR-80-3760, California Institute of Technology, 1980.

[27] E Horowitz and A Zorat. The binary tree as an interconnection network: Applications to multiprocessor systems and VLSI. *IEEE Trans. on Computers*, C-30(4):247–253, 1981.

[28] S Joshi, S Deiss, M Arnold, Jongkil Park, T Yu, and G Cauwenberghs. Scalable event routing in hierarchical neural array architecture with global synaptic connectivity. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on*, pages 1 –6, feb. 2010.

[29] A Joubert, B Belhadj, O Temam, and R Heliot. Hardware spiking neurons design: Analog or digital? In *Proc. 2012 Intl. Joint Conf. Neural Networks (IJCNN)*, June 2012.

[30] F Samman, T Hollstein, and M Glesner. Adaptive and deadlock-free tree-based multicast routing for networks-on-chip. *IEEE Trans. on VLSI*, 18(7):1067–80, 2010.

[31] W Dally and C Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, C-36(5):547–53,

1987.
[32] A Martin. *Programming in VLSI: from Communicating Processes to Delay-Insensitive Circuits*, pages 1–64. UT Year of Programming Series. Addison-Wesley, 1990.
[33] A Lines. *Pipelined asynchronous circuits*. Masters thesis, California Institute of Technology, 1998.
[34] A Martin and Nystrom M. Asynchronous techniques for system-on-chip design. *Proc. of the IEEE*, 94(6):1089–120, 2006.
[35] A Chandrasekaran and K Boahen. A 1-change-in-4 delay-insensitive interchip link. In *Proc. IEEE Intl. Symp. Circ. and Sys.*, pages 3216–19, Piscataway NJ, May 2010. IEEE Press.
[36] R Silver, K Boahen, S Grillner, N Kopell, and K Olsen. Neurotech for neuroscience: unifying concepts, organizing principles, and emerging tools. *The Journal of Neuroscience*, 27(44):11807–11819, 2007.
[37] J Lin and K Boahen. A delay-insensitive address-event link. In *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*, pages 55–62. IEEE, 2009.
[38] L Kleinrock. *Queueing Systems*. Wiley, New York NY, 1976.
[39] S Choudhary, S Sloan, S Fok, A Neckar, E Trautmann, P Gao, T Stewart, C Eliasmith, and K Boahen. Silicon neurons that compute. In *Artificial Neural Networks and Machine Learning–ICANN 2012*, pages 121–128. Springer, 2012.
[40] C Zamarreno-Ramos, A Linares-Barranco, T Serrano-Gotarredona, and B Linares-Barranco. Multicasting mesh aer: A scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. *Biomedical Circuits and Systems, IEEE Transactions on*, PP(99):1, 2012.
[41] S Furber, D Lester, L Plana, J Garside, E Painkras, S Temple, and A Brown. Overview of the spinnaker system architecture. *Computers, IEEE Transactions on*, PP(99):1, 2012.
[42] S Scholze, S Schiefer, J Partzsch, S Hartmann, C Mayr, S Höppner, H Eisenreich, S Henker, B Vogginger, and R Schüffny. Vlsi implementation of a 2.8 gevent/s packet based aer interface with routing and event sorting functionality. *Frontiers in Neuroscience*, 5(117), 2011.
[43] Jongkil Park, T Yu, C Maier, S Joshi, and G Cauwenberghs. Live demonstration: Hierarchical address-event routing architecture for reconfigurable large scale neuromorphic systems. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 707 –711, may 2012.
[44] P Gao, B Benjamin, and K Boahen. Dynamical system guided mapping of quantitative neuronal models onto neuromorphic hardware. *IEEE Trans. on Circ. and Sys.*, 59(10):2383–2394, 2012.
[45] B Benjamin, J Authur, P Gao, P Merolla, and K Boahen. A superposable silicon synapse with programmable reversal potential. In *Proc. Intl. Conf. IEEE Engineering and Medicine in Biology Society (EMBC)*, August 2012.

Paul Merolla's research is to build more intelligent computers, drawing inspiration from neuroscience, neural networks, and machine learning. Paul received his B.S. with *high distinction* in electrical engineering from the University of Virginia in 2000, and his Ph.D in bioengineering from the University of Pennsylvania in 2006. He was a Post-Doctoral Scholar in the Brains in Silicon Lab at Stanford University (2006-2009) working as a lead chip designer on Neurogrid, an affordable supercomputer for neuroscientists. Starting in 2010, Paul has been a research staff member at IBM Almaden Research Center, where he was a lead chip designer for the first fully digital neurosynaptic core as part of the DARPA-funded SyNAPSE project.

Paul's interests includes low-power neuromorphic systems, asynchronous circuit design, large-scale modeling of cortical networks, statistical mechanics, machine learning, and probabilistic computing.

John V. Arthur received the B.S.E. degree (*summa cum laude*) in electrical engineering from Arizona State University, Tempe, and the Ph.D. degree in bioengineering from the University of Pennsylvania, Philadelphia, in 2000 and 2006, respectively. He was a Post-Doctoral Scholar in bioengineering at Stanford University, Stanford, CA, as a lead on the Neurogrid project.

He is currently a research staff member at IBM Almaden Research working on the SyNAPSE project. His research interests include dynamical systems, neuromorphic and neurosynaptic architecture, and hardware aware algorithm design.

Rodrigo Alvarez-Icaza received the B.S. degree in mechanical and electrical engineering from Universidad Iberoamericana, Mexico City, a M.S. degree in Bioengineering from the University of Pennsylvania, and a Ph.D. degree in Bioengineering from Stanford University in 1999, 2005 and 2010 respectively.

Rodrigo is currently a research staff member at IBM Almaden Research Center where his research focuses on brain-inspired computers.

Jean-Marie Bussat (M'00) was born in Annecy, France. He received the M.Sc. degree in electrical engineering from the ESIGLEC, Rouen, France, in 1995 and the Ph.D. degree in electrical engineering from the University of Paris XI, Orsay, France in 1998. He joined the technical staff of the department of physics of Princeton University, New Jersey in 1998 to work on the readout of the electromagnetic calorimeter of the Compact Muon Solenoid (CMS) experiment at CERN, Geneva, Switzerland. He joined the engineering division of the Lawrence Berkeley National Laboratory in 2001 to design instrumentation for physics and material science experiments. In 2007, he joined the Brains in Silicon Lab at Stanford University, Stanford, California to work on Neurogrid.

Since 2009, Jean-Marie has been with Apple Inc., Cupertino, California where he is currently an engineering manager in the Sensing Hardware division.

Kwabena Boahen (M'89, SM'13) received the B.S. and M.S.E. degrees in electrical and computer engineering from the Johns Hopkins University, Baltimore, MD, both in 1989 and the Ph.D. degree in computation and neural systems from the California Institute of Technology, Pasadena, in 1997. He was on the bioengineering faculty of the University of Pennsylvania from 1997 to 2005, where he held the first Skirkanich Term Junior Chair. He is presently an Associate Professor in the Bioengineering Department of Stanford University. He directs Stanfords *Brains in Silicon* Laboratory, which develops silicon chips that emulate the way neurons compute. His contributions include a silicon retina that could be used to give the blind sight, a self-organizing chip that emulates the way the developing brain wires itself up, and a specialized hardware platform (Neurogrid) that simulates a million cortical neurons in real-time while consuming only a few watts. His scholarship is widely recognized, with over eighty publications to his name, including a cover story in the May 2005 issue of Scientific American. He has received several distinguished honors, including a Fellowship from the Packard Foundation (1999), a CAREER award from the National Science Foundation (2001), a Young Investigator Award from the Office of Naval Research (2002), the National Institutes of Health Directors Pioneer Award (2006), and the National Institutes of Health Directors Transformative Research Award (2011).