Address-Event Communication Using Token-Ring Mutual Exclusion

Nabil Imam and Rajit Manohar
Computer Systems Laboratory, School of Electrical and Computer Engineering
Cornell University, Ithaca, NY 14853, U.S.A.
{ni49, rajit}@csl.cornell.edu

***Abstract*—We present a novel Address-Event Representation (AER) transmitter circuit to communicate pulses of neural activity (spikes) within a neuromorphic system. AER circuits allow an ensemble of neurons to achieve large scale time-multiplexed connectivity through a shared communication channel. Our design makes use of token-ring mutual exclusion where two circulating tokens in a 2D array of neurons provide exclusive access to the shared channel. Compared to traditional arbitration-tree-based designs, our design has a higher throughput and lower latency during high spiking activity. This allows the circuit to serve larger neuronal densities and higher spiking activity while maintaining the temporal precision required by the system. Our design also eliminates the address line loads that restricted scalability in previous designs. In addition, our simpler circuit topology leads to area and power savings.**

I. Introduction

VLSI implementations of large scale biologically inspired neural networks has been a topic of widespread interest in the last two decades. Such neuromorphic systems [1] contain models of biological neurons as their basic computational units, massively interconnected to replicate the parallel and distributed nature of biological computation [2]. These chips are both tools for biological investigations as well as platforms for implementing machine perception tasks.

Neurons communicate through pulses of activity called spikes, and the asynchronous generation and modulation of spike trains enable a network of neurons to operate efficiently and reliably under energy constraints. In neuromorphic systems, biology is mimicked by using continous time analog circuit implementations of neurons, and digital asynchronous circuits for spike communication.

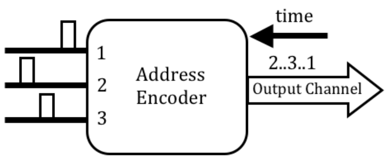


Fig. 1. Time-multimplexed connections through an AER transmitter. Events are encoded by their address and communicated through a shared output channel

The interconnect resources available in semiconductor technologies are much lower than what is required to achieve the thousands of connections that each neuron may have. To overcome this problem, the address-event representation (AER) has emerged as a standard. AER uses time-division multiplexing to achieve large connectivity, using bandwidths of hundreds of megahertz available in semiconductor technologies to serially transmit information between thousands of neurons operating at tens of hertz. In an AER-based system, an array of neurons share an output bus to transmit spikes that are routed to a receiver. Fig. 1 illustrates how the transmitter time multiplexes spike communication. Several methods to share the output bus have been proposed, and their performances have been compared [3]. In this paper we present a new AER transmitter circuit based on a token-ring mutual exclusion architecture. A row token and a column token circulate in a two dimensional array of neurons, and a neuron gains exclusive access to the output bus only when it has both the row and the column tokens. A counter keeps track of the tokens and sends out the row and column token values whenever there is a spike. Depending on the spiking activity, our circuit may either scan a certain section of the array or circulate the token to different parts of the array to provide bus access. This design shows significant performance improvements over previous ones, and leads to much simpler and therefore, more efficient circuits.

In Section II we discuss some previously designed AER transmitters and the trade-offs involved. In Section III we point out that the AER can be viewed as a system implementing distributed mutual exclusion, and discuss several implementation strategies as well as our proposed architecture. Section IV discusses circuit implementation. Simulation results and comparisons with previous designs are provided in Section V, and we conclude in Section VI.

II. Design Trade-offs

Asynchrnous AER communication is preferred in neuromorphic systems since neurons may spend a considerable amount of time being idle and may spike at spontaneous instances. A clocked approach is not well suited for fast servicing of these irregular spike patterns and it also leads to unnecessary power consumption when the neurons are idle. A clock-gated approach may mitigate the idle power consumption, but leads to added latencies in spike communication.

One of the performance criteria for an AER transmitter is the capacity of the output channel. This is defined as the maximum rate at which spikes can be communicated through the channel. The information coding strategy used affects the capacity. Due to increasing neurobiological evidence for spike-time dependent coding [4], we consider the use of fixed height fixed width spikes that temporally encode information. While transmitting this information over a shared channel, latencies are added that reduce timing precision. A neuromorphic system may require the latencies to be bounded as low as possible to maximize timing precision, or it may require the latencies to be kept below the timing precision being used to synchronize the system. The standard deviation of the latencies, known as the temporal dispersion, is also a factor to be considered since it indicates variability between individual neurons. Another design consideration is to maximize the fraction of accurately transmitted spikes. Since spiking activity can overlap in time, there is an option of either queueing overlapping spikes or discarding all but one. Queuing results in further latencies while discarding spikes leads to information loss that is measured as error-rates. The utilization of the channel capacity, defined as the throughput of the channel, is lowered by bounds on these latencies and error-rates. For given bounds, a design with a higher throughput will result in an AER transmitter that can handle larger loads.

One way to design an AER transmitter is to have a predetermined allocation of output channel access times for each of the neurons. This scanning architecture may be implemented synchronously or asynchronously with an external counter, with neurons being sampled one by one at a frequency that is dependent on the size of the population and the latency bounds. Instead of a static allocation of channel access times, the scanning rate can also be dynamic, with more active neurons having a larger share of the channel. When the activity in the neuron array is high, this method of channel access will lead to good throughput since most of the neurons will make use of the channel during their allocated time. However, scanning is not the best solution if the activity in the array is temporally and physically sparse since unnecessary latencies and power consumption are added. In particular, scanning requires inspecting a neuron's state even when the neuron does not have any spikes to communicate.

Rather than having specific time-slots for output channel access, the transmitter can be designed to give neurons unfettered access. The ALOHA-based design [3][5] allows each neuron to access the channel as soon as an event takes place. Events overlapping in time are discarded and this results in a trade-off between high error rates and limited throughput for large array sizes. Improvements on these limitations can be achieved by variations of the ALOHA-based design such as the slotted-ALOHA protocol, where events are allowed output channel access only on discrete-time slots, and the priority encoder algorithm, where cells are assigned a priority, and overlapping events are sent according to the priority of their respective cells.

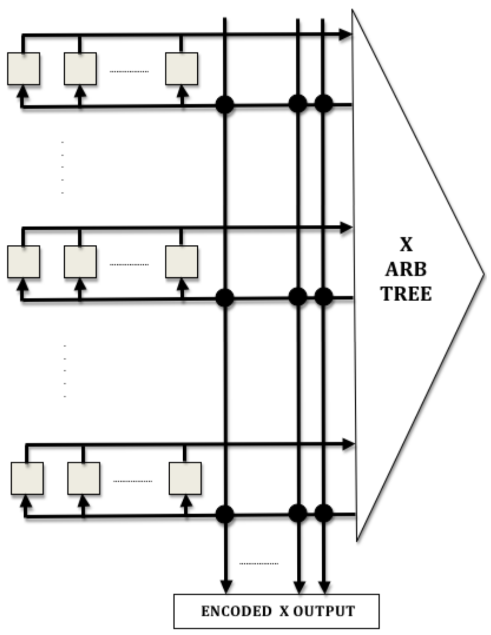


Fig. 2. AER design using an arbitration tree. Overlapping row requests go through log(N) stages of arbitration, and are served one by one. For simplicity, the figure only illustrates row arbitration. Column arbitration is carried out in an identical manner

Another approach to transmitter design is the use of an arbitration tree to queue temporally overlapping events [6]. Implementations of this approach arrange the neurons in two dimensions in order to exploit locality of spiking activity. When a neuron spikes it asserts a row request line that is shared by other neurons in the same row. If multiple rows have temporally overlapping events, then an arbitration tree carries out communication with only one of the rows while the others wait. Neurons of the row that won arbitration then assert their column request lines, and a column arbitration tree communicates with only one of the columns while the others wait. The row and column acknowledge lines are encoded, and the address of the spiking neuron is sent out as the output address. The row arbitration process is illustrated in Fig. 2. Since rows and columns that lose arbitration have to wait till they eventually win, latencies are added to the spike times. This is usually acceptable, since the latency introduced by multiplexing is low enough (μs) compared to the delay between individual spikes (ms). Each neuron has a period of inactivity after they spike. This 'refractory period' (ms), is much larger than the time it takes to service entire neuron arrays. This ensures that all the neurons eventually gets served by the AER.

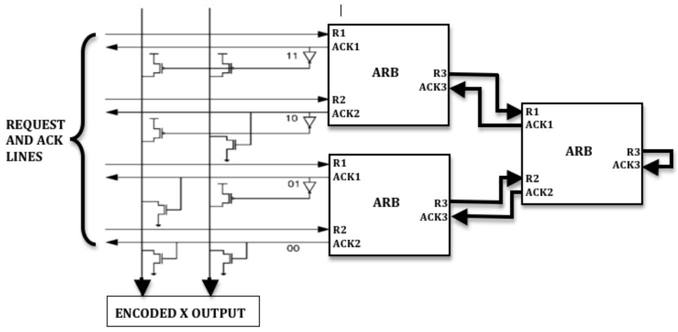


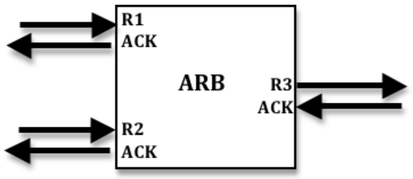
Fig. 3. Encoding of the ack lines for N=2. Each acknowledge line drives 2 transistors, each of which is connected to 3 other transistor drains. As N increases the load on these output lines limit the scalability of the system

Surplus channel capacity is required to minimize the error rates in the unfettered design and the queuing time in the arbitrated one. For the spike patterns seen in neural systems, the required surplus capacity is much smaller in the arbitration-based AER implementations [7]. Therefore the tree-based design has become the standard for most neuromorphic systems. Several modifications to the original circuit have been suggested to improve its performance.

The complexity of address encoding circuits for a large number of neurons is a problem in the tree-based design. Encoding the event address using the row and column acknowledge lines leads to large loads on the output. If the row address is encoded in N bits, then each row acknowledge line will drive N transistor gates, where each transistor is connected to $2^N - 1$ other drains. The switching speed degrades and the energy per event increases with N. Although these drawbacks can be reduced by encoding the address at different stages of the arbitration tree [8] they eventually restrict the scalability of the design. Fig 3 illustrates the encoding required for N=2.

Another problem with the tree-based approach is the potential delay incurred while a spike waits to win arbitration. If there is a spatiotemporal burst of spikes, e.g if all the column of one row spike together, each column of size N has to undergo $O(log(N))$ stages of arbitration per spike, leading to large latencies. This is another factor that limits the scalability of the array. These latencies can be reduced if the arbitration sub-processes are replaced with their greedy counterparts. In the greedy scheme, Fig. 4, each level of the arbitration tree checks if there is a pending communication on the other input channel before completing the second R3 action. If we are to assume that spikes are being generated asynchronously and that a request line may be asserted at any spontaneous time, then checking the value of the probe of R1 and R2 involves negated probes [9], and the greedy process requires two more arbiters and additional circuit complexity for a robust implementation.

Further enhancements to improve the performance of the tree-based approach has been suggested. The design in [10] reads all the columns of a row together and initiates the next cycle of row arbitration while the column addresses are being sent out. The increase in throughput comes at the expense of added circuit complexity. Pipelining the receiver [7] that is reading the output channel can also boost throughput. Merging the row and the column outputs and sequentially sending the row followed by the column addresses in one channel cuts down the required number of output pads for a chip-level AER.



$$
\begin{aligned}
NON-GREEDY: \quad & *[[\overline{R1} \rightarrow R3;R1;R3 \\
& \;\; | \; \overline{R2} \rightarrow R3;R2;R3 \\
&]]
\end{aligned}
$$

$$
\begin{aligned}
GREEDY: \quad & *[[\overline{R1} \rightarrow R3;R1;[\overline{R2} \rightarrow R2 \mid \neg\overline{R2} \rightarrow skip];R3 \\
& \;\; | \; \overline{R2} \rightarrow R3;R2;[\overline{R1} \rightarrow R1 \mid \neg\overline{R1} \rightarrow skip];R3 \\
&]]
\end{aligned}
$$

Fig. 4. Arbitration tree units and their greedy counterparts. If spiking activity is clustered in space and time, latencies can be reduced by acknowledging neighboring requests in sequence through the greedy arbitration process. However, the negated probe leads to extra circuit complexity

Populations of neurons involved in sensory coding (such as in the retina and the cochlea) often have small fractions in physical proximity fire together or in synchrony while the total population activity stays low. In neuromorphic chips that implement sensory function, it is therefore desirable to have an AER that can scan the active neurons without having to scan through the entire array. This can eliminate the latencies that incur when a spike waits to win arbitration. The greedy arbitration tree does this at the expense of additional complexity in the arbitration tree units, leading to area and power penalties. In the next two sections, we describe an AER transmitter design that can scan sections of the array that spike together without incurring extra circuit overhead. In addition, the AER keeps track of where the spiking activity is, thereby eliminating the need to encode the acknowledge lines.

III. Mutual Exclusion and AER

Selecting a neuron row/column, and then selecting an individual neuron within that row/column is simply a special case of implementing the classic problem of distributed mutual exclusion [11]. There are a number of ways to implement such a protocol. The arbiter tree approach is simply a special case where rows and columns are subdivided into groups of size 2, 4, 8 and so on in subsequent levels of the tree, and a 2-way mutual exclusion is carried out between groups.

If we consider the array of neurons as a ring of processes sharing a resource (the output channel), we can use one of several algorithms [12] to implement reliable and efficient AER transmitter circuits. These algorithms implement mutual exclusion by giving one of several masters exclusive access to a shared resource via private servers. The servers communicate with each other to pass a token around, and access to the shared resource can only be provided by a server that has the token. The procedure is illustrated in Fig. 5.

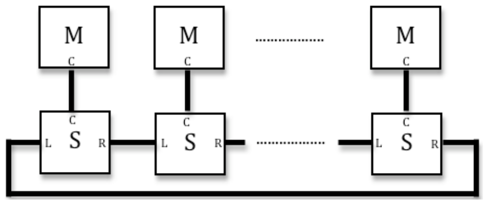


Fig. 5. Mutual exclusion on a ring of processes: A set of masters (M) share a common resource. Each master has its own server (S) and a circulating token among the servers grants exclusive access to the resource

The servers may be designed to circulate the token in various ways. A simple approach is to circulate the token continuously through the servers, sampling each of the masters one by one. This corresponds to the scanning method mentioned in Section II. Another method is to move the token upon request. When a master wants access to the resource, it communicates with its server. The server either grants access if it has the token or asks one of its neighboring servers for the token. The neighboring server either passes the token if it has it or communicates with the next neighbor. A token thus circulates unidirectionally in the server ring, giving exclusive access to the shared resource. A modification to this approach is to allow the token to travel in both directions. The servers would have to keep track of which direction the token last went, and make requests only in that direction to avoid deadlock.

The rows of a neuron array can be considered as a set of masters that request access to a shared channel whenever any neuron in the row spikes. A set of row servers and a circulating row token select spiking rows exclusively. Similarly, the columns of the array can be considered as a separate set of masters. Following the selection of a row, a set of column servers and a circulating column token select spiking columns exclusively. A neuron in the array gains exclusive access to the output channel when both its row and column servers get the tokens.

We chose unidirectional token circulation for our architecture since it leads to the most efficient implementation to handle common spiking activity. Our design is illustrated in Fig. 6. When a neuron spikes it asserts an open-drain row request line to communicate with a row server. If the row server has the row token, it acknowledges the request. Otherwise it sends a communication request to the row server directly below it. The latter server acknowledges this request if it has the row token, thereby given it up. Otherwise it communicates with the server below it and waits for the acknowledge before acknowledging the original communication. While the token is passed, a N-bit counter that holds the row token value is also incremented. This value always corresponds to the row server that currently holds the token. If a row server has pending communication from both its row and its neighboring server, it arbitrates between the requests. Since only the row server that has the token can acknowledge a row request, spiking rows are selected exclusively. Once a neuron gets its row request acknowledged, it asserts a column request line to communicate with its column server. The column servers have a similar operation as the row servers. When the servers pass the column token they assert an increment line that updates the column count in the counter. Once a server with a pending column request gets the token, it carries out a second communication with the counter to send out the row and the column count on the output channel.

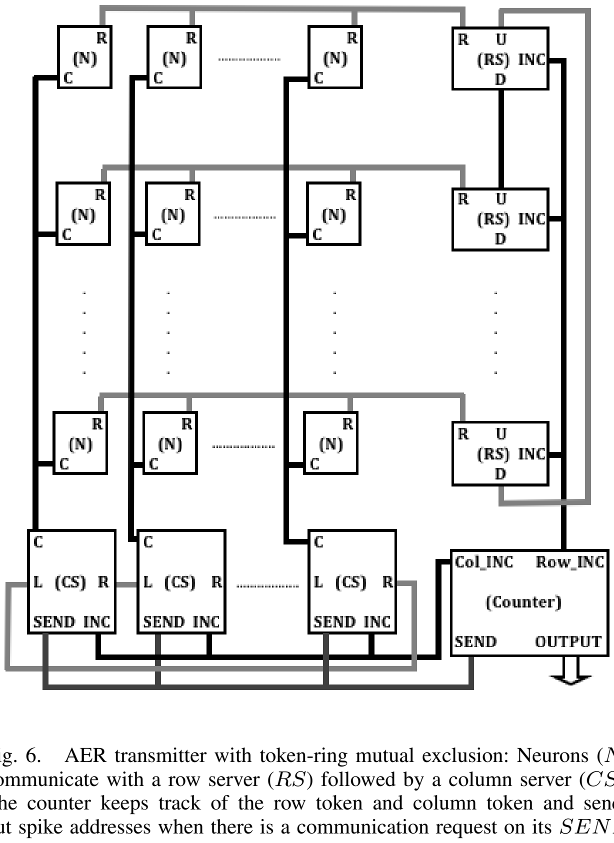


Fig. 6. AER transmitter with token-ring mutual exclusion: Neurons (N) communicate with a row server (RS) followed by a column server (CS). The counter keeps track of the row token and column token and sends out spike addresses when there is a communication request on its $SEND$ channel

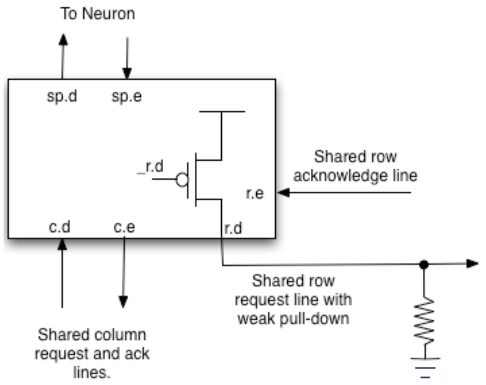
IV. Circuit Implementation

We describe the concurrent processes of our architecture using Communicating Hardware Processes (CHP) and construct them using Martin's synthesis method [13].

A. Neuron Interface

The process that communicates with a row server and a column server is the interface between a neuron and the AER transmitter. Fig. 7 shows a block diagram for the process and its CHP is:

```
$S \equiv$
    $*[[\overline{S} \longrightarrow R!; C!; R!; S?]]$
```



To Neuron
sp.d
sp.e
_r.d
Shared row acknowledge line
r.e
c.d
c.e
r.d
Shared row request line with weak pull-down
Shared column request and ack lines.

Fig. 7. The neuron interface process. The row request line is open-drain in order to avoid interference while the neurons in the row are being served. A weak-pull down transistor pulls the line down once all the neurons release the line

S is the communication channel with the neuron. When there is a spike, the process carries out a two-phase handshake with a row server through the R channel. This channel is shared by all the neurons in the same row. Following this communication, the process communicates with a column server through the C channel. This communication completes after the column token has arrived and the counts have been sent to the output channel. A second communication with the R channel indicates that the row token is no longer needed by the neuron. The handshaking expansion of the process is given below. We use signals ending in ".e" to refer to the inverted sense of the acknowledge (the *enable*), and signals ending in ".d" to refer to the request (data).

```
$*[[s.d]; \_r.d\downarrow; [\neg r.e]; c.d\uparrow; [\neg c.e];$
$s.e\downarrow;\ c.d\downarrow;\ [c.e];\ \_r.d\uparrow; [r.e][\neg s.d];\ s.e\uparrow]$
```

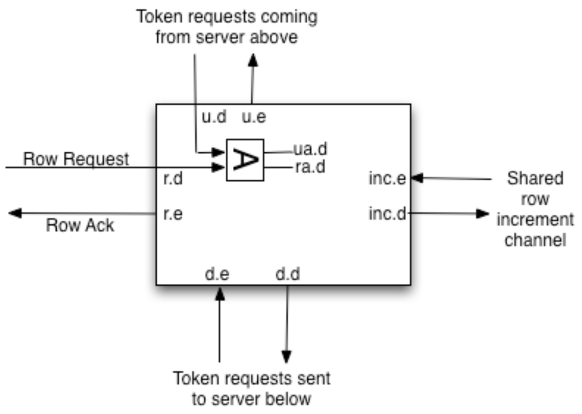
The R channel has an open drain request line so that neurons in the same row do not attempt to pull the line towards opposite voltages at the same time (interference). Spiking neurons sharing a row request line can only pull up the line (through their local signal _r.d) and they release it after communication with their respective column servers has finished. When all the neurons release the line, a weak-pull down transistor drives it low.

Since R communication occurs twice it is implemented as a two phase handshake. The column server communication is implemented as a four phase handshake, and the acknowledge signal back to the neuron is used to distinguish the states between this handshake.

B. Row Server

The row server is illustrated in Fig. 8 and its CHP is given below:

```
$RS \equiv$
    $*[[\overline{R} \longrightarrow [b \longrightarrow skip\ [] \neg b \longrightarrow D!; b\uparrow\ ]; R?; R?$
     $|\overline{U} \longrightarrow [b \longrightarrow skip\ [] \neg b \longrightarrow D!\ ]; INC!; b\downarrow; U?$
    $]]$
```



Token requests coming from server above
u.d
u.e
ua.d
ra.d
Row Request
r.d
Row Ack
r.e
inc.e
inc.d
Shared row increment channel
d.e
d.d
Token requests sent to server below

Fig. 8. Block diagram of the row server. Simultaneous requests from the row and a neighboring server are arbitrated. The INC channel goes to the counter and is shared by all the other row servers

The process waits for a communication request from either a row (through the R channel) or the server above (through the U channel). If both communicate together, the signals are arbitrated. If the row has sent a request and has won arbitration, the RS process acknowledges with a two phase handshake if the local variable b, representing the row token, is high. If the server doesn't have the row token, b is false and a communication request is sent to the row server below. Upon acknowledgement, the process updates b and acknowleges R. The second two phase handshake on R is completed once the row request line has gone down, i.e. after all the pending requests from the row have been served. The handshaking expansion we used was:

```
$*[[r.d \longrightarrow$
    $[b \longrightarrow skip$
    $[] \neg b \longrightarrow d.d\uparrow;\ [\neg d.e];\ b\uparrow;\ d.d\downarrow;\ [d.e]$
    $];\ r.e\downarrow;\ [\neg r.d];\ r.e\uparrow$
  $|u.d \longrightarrow$
    $[b \longrightarrow skip$
    $[]\ \neg b \longrightarrow d.d\uparrow;\ [\neg d.e];\ b\uparrow;\ d.d\downarrow;\ [d.e]$
    $];\ up.e\downarrow;\ inc.d\uparrow;\ [\neg inc.e];\ b\downarrow;\ inc.d\downarrow;$
    $[inc.e];\ [\neg up.d];\ up.e\uparrow$
$]]$
```

While reshuffling[14], we had to take into account the fact that the increment line going to the counter is shared by all the row servers. The full four-phase handshake of D needs to finish before a server asserts the shared line in order to avoid interference. To reduce the load on the shared line, we minimized the guards required to drive the $inc.d$ line. While buffers can reduce this load, they add extra transitions that slow down token movement.

During reshuffling, proper sequencing with respect to the rest of the system needs to be kept in mind. For example, in order to minimize latency, the row acknowledge line r.e can be asserted without waiting for the full four phase handshake with the D channel to finish. However, this would mean that the neurons may initiate column communication before the row token increment cycle has been initiated. Depending on the position of the neuron in the array, and the position of the column token, the column servers may initiate a send cycle before the row increment cycle starts. While this is an unlikely scenario, the shared wires may be quite long, and it is best to avoid such timing assumptions. Therefore r.e is pulled down only after the full handshake with D ends, i.e only after the row increment cycle has completed.

The reshuffling used also avoided extra state variables thus minimizing the number of transitions required for servicing a request. The shared inc.d line that is driven by all the row servers is state-holding. We add one shared staticizer outside the row servers. Depending on the size of the array, this staticizer may be divided up into equal pieces and distributed across the line.

Note that the sequence $[\neg r.d]; r.e\uparrow$ leads to a potentially unstable production rule. If a spike is asserted on the row request line at the time the transition $r.e\uparrow$ is taking place, there will be a glitch in the circuit. A related sequence of events in the neuron interface handshake is $[s.d]; \_r.d\downarrow; [\neg r.e]; c.d\uparrow$. If a spike occurs while the long $r.e$ wire is switching up, the column request line may be asserted even though the corresponding row server may have given up its token. These potentially unstable scenarios are also present in the arbitration-tree-based transmitters since they share the row request lines in an identical way. We propose two methods to tackle this problem.

One solution is to introduce a 'wait' command in the neuron interface. This wait is added between the initial check of $[\neg r.e]$ and the pull-up of the column request line. The wait can be implemented with a simple delay line, with the delay being equal to the time it takes for the entire r.e line to be pulled up. After this wait, the r.e line is checked a second time to confirm that it is low before proceeding. If a neuron pulls up the row request line while it is on its way down, there may be one of two scenarios:

- On the server side, if the request line has gone down low enough, it will lose arbitration with a high u.d (neighbor token request line) and r.e will go high. On the neuron side, the wait command followed by the second check of r.e will prevent the handshake from proceeding.
- On the server side, if the request line has not gone low enough to lose arbitration, or if the u.d signal is low, then the pull up of r.e will be prevented. In the neuron side, the column request line will be pulled up after the second check for a low r.e.

The modification to the neuron interface HSE is shown below:

```
*[[$s.d$]; $\_r.d\downarrow$; [$\neg r.e$]; $delay\_start\uparrow$; [$delay\_end$];
[$\neg r.e$]; $c.d\uparrow$; [$\neg c.e$]; $s.e\downarrow$; $c.d\downarrow$; [$c.e$]; $\_r.d\uparrow$;
[$r.e$], [$\neg s.d$]; $s.e\uparrow$]
```

The length of the wait is only the time required to switch one long wire (a few ns at most). The wait only affects the service time of the first spike serviced in the row. The wait of the other spikes proceed (and complete) while some spike in the row is being serviced. Thus the addition of this wait has a negligible change in the performance of the AER.

An alternate way of solving the glitch problem is to synchronize the system such that all neurons receive their synapses and evaluate their membrame potentials in the same cycle, and spikes are sent out in the next cycle. This will ensure that all neurons pull up their row request lines within a time window that will avoid the unstable situation.

C. Column Server

The column server is illustrated in Fig. 9 and its CHP is given below:

```
$CS \equiv$
    *[[$\overline{C}$ $\longrightarrow$ [$b \longrightarrow skip$ ▯$\neg b \longrightarrow L!; b\uparrow$ ];
                    $C?; SEND!; C?$
     |$\overline{R}$ $\longrightarrow$ [$b \longrightarrow skip$ ▯$\neg b \longrightarrow L!$ ]; $INC!; b\downarrow; R?$
    ]]
```

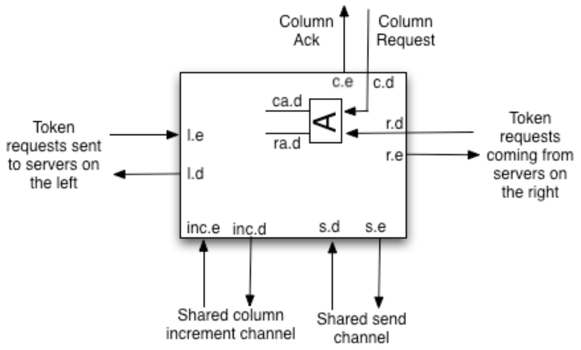

Column Ack
Column Request
c.e
c.d
ca.d
A
ra.d
r.d
Token requests sent to servers on the left
l.e
l.d
r.e
Token requests coming from servers on the right
inc.e
inc.d
s.d
s.e
Shared column increment channel
Shared send channel

Fig. 9. Block diagram of the column server. Simultaneous requests from the column and a neighboring server are arbitrated. The INC and $SEND$ channels communicate with the counter and are shared by all the other column servers. There is a shared staticizer added to the incoming column request line

The CHP is identical to that of the row server except for an extra active communication action $SEND$ that prompts the counter to send out the row and column counts to the output channel. In addition, unlike the row requeset lines, the column request lines are not open-drain and do not have weak pull-downs. Since the lines are state-holding, a shared staticizer is used.

The handshaking expansion that was used was:

```
$*[[c.d \longrightarrow [b \longrightarrow skip$
                    $[]\neg b \longrightarrow l.d\uparrow;\ [\neg l.e];\ b\uparrow;$
                                   $l.d\downarrow;\ [l.e]$
                    $];\ c.e\downarrow;\ s.d\uparrow;\ [\neg s.e];\ [\neg c.d];$
                    $c.e\uparrow;\ s.d\downarrow;\ [s.e]$
   $|r.d \longrightarrow [b \longrightarrow skip$
                   $[]\neg b \longrightarrow l.d\uparrow;\ [\neg l.e];\ b\uparrow;$
                                  $l.d\downarrow;\ [l.e]$
                   $];\ r.e\downarrow;\ inc.d\uparrow;\ [\neg inc.e];\ b\downarrow;$
                   $inc.d\downarrow;\ [inc.e];\ [\neg r.d];\ r.e\uparrow$
   $]]$
```

Similar to the row server case, we reshuffled the column server handshaking expansion in order to minimize the load on the shared channels INC and $SEND$, as well as to avoid extra state variables and to maintain proper sequencing with respect to the rest of the system.

D. Counter

Fig. 10 shows a block diagram of the modulo N counter. Its CHP is:

```
$COUNTER \equiv$
   $*[[\overline{INC\_R} \longrightarrow rct := rct + 1;\ INC\_R?$
     $[]\overline{INC\_C} \longrightarrow cct := cct + 1;\ INC\_C?$
     $[]\overline{SEND} \longrightarrow B!(cct, rct);\ SEND?$
   $]]$
```

B is the $log(N \times N)$ output channel. The local variables rct and cct are $log(N)$ bits each and represent the row and column counts respectively. $B!(cct, rct)$ represents both the counts being sent to the output channel.

We can decompose the CHP as follows:

```
$CONTROl \equiv$
   $*[[\overline{INC\_R} \longrightarrow ifb := rct;\ [v(ofb)];\ rct := ofb;$
                          $ifb \Downarrow;\ [n(ofb)];\ INC\_R?$
     $[]\overline{INC\_C} \longrightarrow ifb := cct;\ [v(ofb)];\ cct := ofb;$
                          $ifb \Downarrow;\ [n(ofb)];\ INC\_C?$
     $[]\overline{SEND} \longrightarrow B!(cct, rct);\ SEND?$
   $]]$
   $\|$
   $*[[v(ifb)];\ ofb := ifb + 1;\ [n(ifb)];\ ofb \Downarrow]$
```

The predicates $v(\cdot)$ and $n(\cdot)$ indicate the validity and neutrality of the delay-insensitive code representing the counter values. The notation $x \Uparrow$ is used to indicate setting the delay-insensitive code for x to the appropriate valid value, and $x \Downarrow$ corresponds to setting the code to a neutral value [15].

The purpose of this decomposition is to use the same increment block for updating both the row and column counts. This is possible since rct and cct are incremented on a mutually exclusive basis. The decomposed design is illustrated in Fig. 10. We used the following handshaking expansion for the control:

```
$*[[inc\_r.d \longrightarrow x\downarrow;\ ifb := rct;\ [v(ofb)];\ rct := ofb;$
                    $inc\_r.e\downarrow;\ [\neg inc\_r.d];\ x\uparrow;\ ifb \Downarrow;$
                    $[n(ofb)];\ inc\_r.e\uparrow$
  $[]inc\_c.d \longrightarrow y\downarrow;\ ifb := cct;\ [v(ofb)];\ cct := ofb;$
                    $inc\_c.e\downarrow;\ [\neg inc\_c.d];\ y\uparrow;\ ifb \Downarrow;$
                    $[n(ofb)];\ inc\_c.e\uparrow$
  $[]send.d \longrightarrow w\downarrow;\ b.d \Uparrow;\ [\neg b.e];\ send.e\downarrow;$
                    $[\neg send.d];\ w\uparrow;\ b.d \Downarrow;\ [b.e];$
                    $send.e\uparrow$
$]]$
```

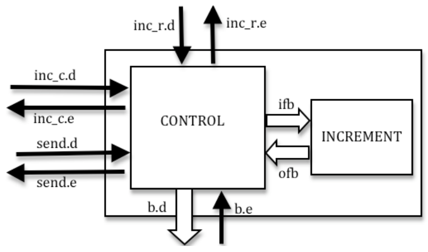


Fig. 10. Block diagram of the counter. The same increment block is used to increment the row and column counts exclusively. Shared staticizers are added to the *inc.d* and *send.d* lines

State variables w, x and y were used to distribute the loads on the increment block input and the address output channel. We can carry out a transformation to increase the performance of the counter. Consider the following sequence of actions:

$$[v(ofb)];\ rct := ofb$$

Instead of waiting for all the bits of ofb to be valid before changing rct, we can transform the expression as follows:

```
   $[(\wedge :: v(ofb_j))];\ (\| j :: rct_j \Uparrow)$
$\triangleright$ $(\| j : [v(ofb)_j]);\ (\| j :: rct_j \Uparrow)$
$\triangleright$ $(\| j : [v(ofb)_j];\ rct_j \Uparrow)$
$\triangleright$ $(\| j : [ofb_j.t \longrightarrow rct_j\uparrow\ []ofb_j.f \longrightarrow rct_j\downarrow\ ])$
```

In the first line we explicitely wrote the handshaking expansion for the sequence of actions. The second line uses the fact that the conjunction of validities is equivalent to waiting for each valid bit. The third line is due to the fact that rct_j only depends on ofb_j. The fourth line explicitely sets the correct value of rct_j. With this transformation, the bits of rct can change as soon as the corresponding bit of ofb has become valid. The case for cct is identical. This sequence of transformations is similar to those used to implement function block style datapaths [15].

The incrementer block is made up of N identical one bit adders, with the carry-in of the first adder always one, and the carry-out of the last adder always ignored.

V. Results and Discussion

The token-ring AER circuits presented in the previous section have been designed and layed out in a 45 nm CMOS SOI process. The AER circuit was designed to service

16x16 neuron arrays on a chip that has been submitted for fabrication. In this section we present pre-layout SPICE simulation results to compare the token-ring architecture with the two basic techniques that are used to design AER transmitters - scanning and tree-based arbitration. All circuits being compared were sized with identical heuristics. Since HSIM/HSPICE pre-layout simulations do not take into account wire capacitances, we added wire loads to every gate in the circuit to account for the large wire capacitances that would be seen in the shared lines. Post-layout simulations indicated that our load estimates are conservative.

In Section II we discussed how neural activity occur in bursts, with spikes grouped together in space and time. In these scenarios, a straightforward arbitration-tree based design consisting of simple arbitrating sub-processes incur large latencies since all spiking units in the neuron array has to undergo $log(N)$ stages of arbitration one by one before being served. In contrast, the token-ring architecture serves all neurons in spatial proximity one by one without the requests having to go through multiple stages of arbitration each time. Since neighboring neurons are scanned sequentially with minimal latency in between, the token-ring structure results in higher throughput. To achieve a desired timing precision, this architecture is able to service a larger neuron array and higher spiking activity. Fig. 11 illustrates how the token-ring AER transmitter results in faster service times compared to a tree-based one in a scenario where all neurons in a row have spiked together.

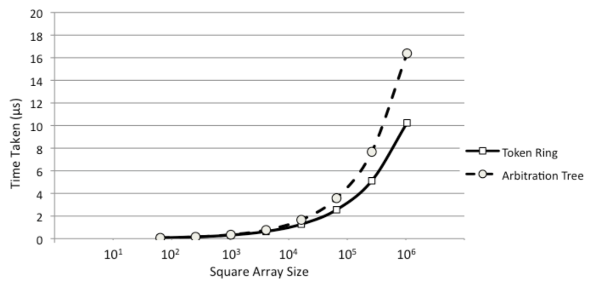


Fig. 11. Service times for different sizes of a square array of neurons. Since spikes occur in spatiotemporal clusters, we consider an entire row of neurons spiking together. Larger array sizes lead to increased latencies in the arbitration tree, and the token-ring scanning results in a distinct speed up in service time

During periods of isolated activity in the neuron array the arbitration tree may lead to lower latencies than the token-ring because a token may have to travel $O(N)$ stages before a spike is processed. The worst-case latency in the token-ring occurs when both the row token and column token need to travel $N-1$ stages before servicing a spike. To service the isolated spike, the tree-based circuit needs to go through the $log(N)$ stages of row and column arbitration only once, whereas the scanning architecture needs to sample N^2-1 neurons in the worst case. The performance of the token-ring is better than a purely scanning architecture since the token simply skips the rows and columns that are inactive.

The added latencies when the token is out of position can be reduced by modifying the servers so that the token can move in both directions. The only modification necessary is the addition of a binary variable in the server processes that is switched on or off depending on which direction the token has previously moved. Requests for the token are made in the appropriate direction.

The case of isolated activity with the token out of position is not a frequent one. The bursty activity that is very characteristic of neural systems will negate the latency losses when the token is out of position. Once the token arrives in the neighborhood of the burst, the fast service time of the spikes will make up for the time lost during token travel. Fig. 12 shows the performance results for a scenario where the token has to travel N-1 row servers to service a burst of activity in one row. Once the token arrives at the correct row, the columns are rapidly scanned to make up for the initial lost time.

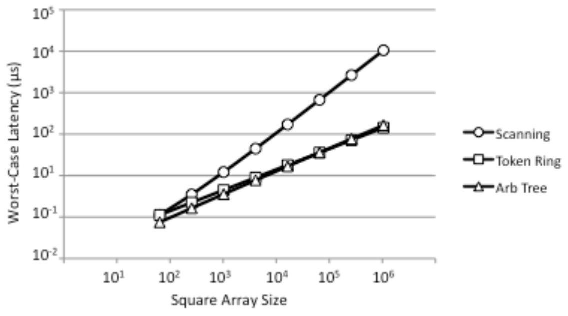


Fig. 12. Worst case latency during bursty activity in a square array of neurons. Even if the token is in the worst-case position, rapid scanning of a burst of spikes makes up for the latencies added due to token travel

For a sufficiently large array size or a sufficiently high spiking rate, the average latency of the token-ring AER is lower than that of the arbitration-tree AER. This is illustrated in Fig. 13. The arbitration tree has a constant latency for a given array size since all spikes need to undergo $log(N)$ stages of arbitration. This latency grows with the size of the array. On the other hand, the average latency of the token-ring is dependent on both the array size and the amount of spiking activity. As the size of the array increases, the average latency increases at a progressively slower rate since the increased time of token travel is balanced by the increased number of spiking neurons that fall in the path of the travelling token. The larger the spiking activity, the lower is the average latency since the length of token travel per spike served goes down.

Since a counter keeps track of the tokens and outputs their positions whenever there is pending communication on its $SEND$ channel, the token-ring architecture eliminates the need for expensive encoding processes. As we discussed in

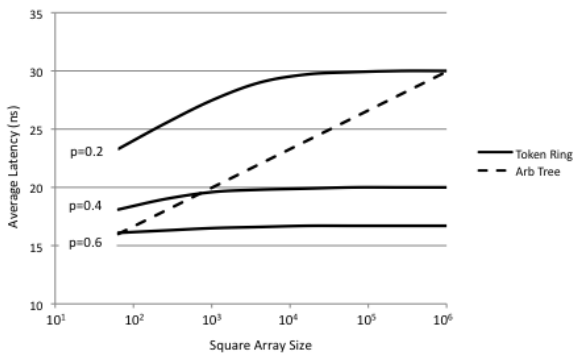


Fig. 13. Average latency per spike for different neuron array sizes and different spike rates. The average latency in the arbitration tree is independent of the spiking activity since every spike needs to undergo $log(N)$ stages of arbitration. Each neuron in the square array was assumed to spike according to a Bernoulli distribution with mean p.

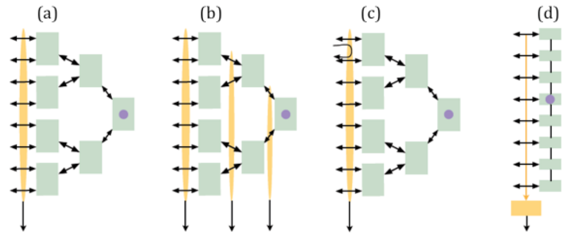


Fig. 14. Comparison of the token-ring AER transmitter with previous designs. (a) The original arbiter tree where each request needs to go through $log(N)$ stages of arbitration before being serviced. Expensive encoding process to set the output lines restrict scalability of the design. (b) Distributed encoding [8] reduces the load on the output lines. (c) A greedy arbitration tree can efficiently service bursts of activity at the expense of circuit complexity. (d) Token-ring mutual exclusion efficiently services bursty activity and eliminates the need for address encoding.

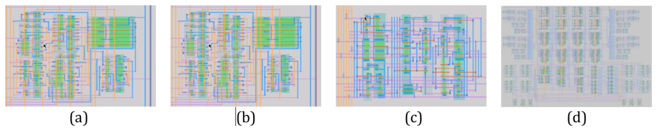


Fig. 15. The simplicity of the token-ring topology leads to area savings and ease of layout. (a) A row server measuring $11x8um^2$. (b) A column server measuring $12x8um^2$. (c) A neuron interface measuring $8x5um^2$. (d) The counter measuring $44x34um^2$

Section II, encoding using the row and column acknowledge lines in the tree-based design leads to output line load capacitances that are proportional to $N * 2^N$, where N is the number of bits required to encode the address of each dimension in the array. The exponential increase in these output capacitances restricts the scalability of the desgin. Using the N-bit counter allows much better scalability. The capacitance in the shared increment lines is proportional to 2^N, but can be easily reduced by splitting the shared wire and recombining through intermediate OR gates

Minus the communication with the counter, the server processes have the same number of communication actions and the same number of arbiters as the arbitration tree units. Thus both processes will have similar power consumptions with regards to internal arbitration and communication with neighbors. For large N values, the counter of the token-ring transmitter has a lower transistor count (proportional to N) than the encoding processes of the tree-based design (2^N transistors on each of $2N$ output wires). Thus we may expect some static power improvements in the token-ring design. The counter also eliminates the large capacitance (propotional to $N{*}2^N$) from $2N$ output wires at the expense of 3 shared wires with lower capacitance (proportional to 2^N, but can be reduced with intermediate OR gates). This will significantly improve the dynamic power consumption for clustered spiking activity since the output wires will be frequently switching. In our implementation of the 16x16 token-ring AER, we found the static power consumption to be 37 uW, and the dynamic power consumption during serving an entire row of spikes to be 236 uW.

An arbitration tree with greedy units can scan neighborhoods of bursty activity to increase the throughput (as described in Section II). However, this comes at the expense of more complicated circuits that include approximately three times as many arbiters due to the necessity of checking potentially unstable communication probes. The token-ring circuit has an inherent ability to scan spike bursts, and does so with reduced circuit complexity. In addition the token-ring servers have a simpler and more compact physical implementation. The layouts of the individual units and their sizes are shown in Fig. 15.

A comparison of the simple token-ring design with previous tree-based designs is illustrated in Fig. 14. The tree-based designs can be interpreted as a token moving along the branches of a tree. In the simple arbitration tree, a token rests on the top of the tree and travels down to the leaves to serve requests. After serving the request, the token travels back to the top. In the greedy tree, the token serves branches at the lower level if they have requests before moving back to the top. Interprating the arbitration tree in this way allows a modification of the designs to allow a counter to replace the expensive encoding processes of the original designs. This could be used in neuromorphic systems that require rapid servicing of isolated spiking activity.

In Section II, we described another enhancement of the AER transmitter that allows arbitration of other rows to start while the column addresses of one row are being sent. This improvement can be easily incorporated in the token-ring design without the high circuit overhead of the original design [10]. In order to gain the extra throughput of this enhancement, the counter is split into a row counter and a column counter. The row server uses an additional communication channel to prompt the row counter to send out the value of the row count to the output channel when there is a row request. The row count is sent only once after which the row token is allowed to move to other servers. Thus the spike receiver will observe a row output from the row counter followed by one or several column outputs from the column counter.

In summary, compared to existing schemes for tree-based AER and scanning, the token-ring based approach is superior in terms of performance during common spiking activity. Compared to greedy arbiter-based AER schemes, the token-ring approach provides equivalent performance at significantly reduced circuit complexity. The only situation where a token-ring based approach is inferior is in the case of an isolated spike in the array that is far away from the current token location; however, this is a case of very sparse spiking activity, and so the additional latency for the spike is not problematic.

VI. Conclusion

Address-Event Representation can be viewed as an implementation of distributed mutual exclusion. We presented an AER transmitter design using a token-ring mutual exclusion protocol. Our token rings handled asynchronous activity in a two-dimensional network of spiking neurons by distributing two tokens across rows and columns of the network and allowing spiking neurons to exclusively access the output channel. Since our design included a counter that keeps track of the token positions, large encoding loads on the output bus were not necessary. Compared to arbitration-tree-based designs, our architecture showed improvements in throughput during bursty spiking activity, thereby allowing more neurons to be serviced within given latency bounds. Our design leads to much simpler, and therefore more efficient, circuits compared to previous designs. We also demonstrated that enhancements can be made to the token ring circuit to further boost performance. Timing assumptions during circuit synthesis were kept at a minimum. Our approach thus resulted in a robust and efficient AER transmitter capable of high-speed communication of bursty spiking activity.

Appendix

The CHP notation we use is based on Hoare's CSP [16]. A full description of CHP and its semantics can be found in [17]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \,[]\, ... \,[]\, Gn \rightarrow Sn]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is $true$, and then executing one of the statements with a $true$ guard. The notation $[G]$ is short-hand for $[G \rightarrow skip]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "$|$" instead of "$[]$."
- Repetition: $*[G1 \rightarrow S1 \,[]\, ... \,[]\, Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the $true$ guards and executing the corresponding statement, repeating this until all guards evaluate to $false$. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is $true$ iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.

References

[1] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, pp. 1629 –1636, Oct. 1990.

[2] C. Mead, *Analog VLSI and neural systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[3] E. Culurciello and A. Andreou, "A comparative study of access topologies for chip-level address-event communication channels," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 1266 – 1277, 2003.

[4] Y. Dan and M.-M. Poo, "Spike timing-dependent plasticity: From synapse to perception," *Physiol Rev 86:1033-1048*, 2006.

[5] A.S.Tanenbaum, *Computer Networks*. Upper Saddle River,NJ: Prentice-Hall, 1996.

[6] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie, "Silicon auditory processors as computer peripherals," *Neural Networks, IEEE Transactions on*, vol. 4, pp. 523 –528, May 1993.

[7] K. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 47, pp. 416 –434, May 2000.

[8] J. Georgiou and A. Andreou, "High-speed, address-encoding arbiter architecture," *Electronics Letters*, vol. 42, no. 3, pp. 170 – 171, 2006.

[9] R. Manohar, M. Nystrom, and A. J. Martin, "Precise exceptions in asynchronous processors," in *Proceedings of the 2001 Conference on Advanced Research in VLSI*, ARVLSI '01, (Washington, DC, USA), pp. 16–, IEEE Computer Society, 2001.

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

[11] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison Wesley Publishing Company, Inc., 1988.

[12] A. J. Martin, "Distributed mutual exclusion on a ring of processes," *Sci. Comput. Program.*, vol. 5, pp. 265–276, October 1985.

[13] A. J. Martin, "Compiling communicating processes into delay-insensitive vlsi circuits," tech. rep., Pasadena, CA, USA, 1986.

[14] R. Manohar, "An analysis of reshuffled handshaking expansions," in *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pp. 96 –105, 2001.

[15] A. J. Martin, "Asynchronous datapaths and the design of an asynchronous adder," tech. rep., Pasadena, CA, USA, 1991.

[16] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[17] A. J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication, UT Year of Programming Series* (C. A. R. Hoare, ed.), pp. 1–64, Addison-Wesley, 1990.