A Burst-Mode Word-Serial Address-Event Link—I: Transmitter Design

Kwabena A. Boahen

***Abstract*—We present a transmitter for a scalable multiple-access inter-chip link that communicates binary activity between two-dimensional arrays fabricated in deep submicrometer CMOS. Transmission is initiated by active cells but cells are not read individually. An entire row is read in parallel; this increases communication capacity with integration density. Access is random but not inequitable. A row is not reread until all those waiting are serviced; this increases parallelism as more of its cells become active in the mean time. Row and column addresses identify active cells but they are not transmitted simultaneously. The row address is followed sequentially by a column address for each active cell; this cuts pad count in half without sacrificing capacity. We synthesized an asynchronous implementation by performing a series of program decompositions, starting from a high-level description. Links using this design have been implemented successfully in three generations of submicrometer CMOS technology.**

***Index Terms*—Asynchronous logic synthesis, event-driven communication, fair arbiter design, neuromorphic systems, parallel readout, pixel-level quantization.**

I. Scaling Two-Dimensional Arrays

Multiple-access inter-chip communication links were originally developed to read out analog signals from sensor arrays. A clock switches the multiplexer from one sensor to another, reading a value from each and every one at a fixed interval, hence the nickname "scanner" [1]. Use of these clock-driven multiplexers continued after quantizers were included in active pixel sensors [2]–[4] and in pulse-coded neural networks [5] to discretize signals inside the array. However, the all-or-none transitions so produced, called **events**, may be output as soon as they occur. Such event-driven access has clear advantages over clock-driven access when activity is sparse (e.g., spatial or temporal filtering occurs) and timing is critical (e.g., time encodes analog information). Consequently, this scheme has been explored for silicon retinas [6]–[9] and silicon cochleas [10].

Several ways of regulating access in event-driven communication systems have been proposed [5], [11]–[13] and their efficiency compared [14], [15], but little attention has been paid to their scaling properties. In all the architectures proposed so far, a single active cell is read, its state is cleared, and then the next cell is read. However, it takes longer to cycle the row and column lines as feature sizes shrink because faster logic (minimum-sized inverter chain) is neutralized by larger load (cells per row or column). Hence, these existing designs cannot accommodate the increase in cell count with integration density—unless the widths of transistors that drive the row and column lines are increased drastically. However, some of these devices actually reside in the cell, which must signal when it becomes active in an event-driven system.

In this paper, we describe a scalable event-driven transmitter interface inspired by two-dimensional (2-D) scanners, which read out an entire row of cells in parallel, over the column lines [1]. The increase in parallelism as the array gets denser enables these analog multiplexers to increase their readout rate, despite the fact that the larger load neutralizes the faster logic. By reading an entire row at once—instead of one cell at a time—we also achieve a transmission rate that increases as the square-root of the cell count, assuming a square array. Such scaling is the best we can do without sizing-up devices inside the array—or breaking it up into separate banks [16]. Our approach requires large devices only in the periphery, where parallel-to-serial conversion occurs. Therefore, it allows designers to take better advantage of higher integration densities offered by advanced submicrometer processes.

Our design uses **address-events** to communicate between cells in the same array or in different arrays, which need not be on the same chip. In this respect, it is similar to previous event-driven links, where the transmitter uses an encoder to generate an address that uniquely identifies an event's place of origin while the receiver uses a decoder to recreate the event at the destination [6], [10], [11]. However, whereas these previous designs transmit row and column addresses in parallel, we transmit them serially. There is no loss in speed because we do not retransmit the row address if the next event is from the same row.

In addition to providing a communication standard for parallel distributed processing, address-events support *virtual* point-to-point connectivity. These virtual wires can be routed by using a look-up table to translate in-coming addresses into one or more out-going addresses [17]–[19]. Furthermore, the single-transmitter–single-receiver link may be extended to support multiple transmitters and receivers using merges and splits [20], or with a shared bus [18], [21]. Thus, the basic link can serve a wide variety of purposes when augmented appropriately. As in previous work, we implemented the link asynchronously to facilitate its use in large heterogeneous multichip systems.

The paper is divided into five sections. In Section II, we briefly review three common multiplexing schemes: metered-,

Manuscript received January 3, 2002; revised November 2002. This work was supported in part by the Whitaker Foundation and in part by the National Science Foundation's LIS/KDI and CAREER programs under Grant ECS98-74463 and Grant ECS00-93851. This paper was recommended by Associate Editor G. Cauwenberghs.

The author is with the Department of Bioengineering, University of Pennsylvania, Philadelphia, PA 19104-6392 USA (e-mail: boahen@seas.upenn.edu).

Digital Object Identifier 10.1109/TCSI.2004.830703

1057-7122/04$20.00 © 2004 IEEE

free-, and arbitered-access (see [15] for an in-depth review). In Section III, we present a high-level specification for the transmitter, and decompose it into a hierarchy of concurrent subprocesses. In Section IV, we present the final handshaking sequences and the resulting asynchronous logic circuits; intermediate synthesis steps can be found in the Appendix. Section V concludes the paper. A preliminary report of this work was presented in [22]. A parallel-write burst-mode receiver and analysis and test results are presented in companion papers [23], [24].

II. Multiplexing Events

Metering access to each cell according to a fixed readout sequence is the simplest solution. These clock-driven multiplexers, or scanners, are commonly used to read out analog currents from imagers, going from row to row, in sequence. In fact, they read all a row's pixels in parallel, and then scan them out serially on the periphery. Thus, they achieve rates over ten million pixels per second; fast enough to scan arrays with hundreds of thousands of pixels at video frame rates. However, these fast analog signals require specialized input/output (I/O) pads, are prone to clock feedthrough and to noise, cannot be easily interfaced with computers, and can be demultiplexed only when array sizes and clock speeds match.

If activity is sparse, it is more efficient to transmit a cell's state only when it changes. It is easy to recognize when significant changes occur if cell-state is quantized. The cheapest quantizers are one-bit analog-to-digital converters, such as integrate-and-fire neurons [2], [8], [25] or sigma-delta encoders [3], [4]. Their fixed-width–fixed-height spikes or binary state-transitions constitute a sequence of events that encode information only in their timing. When coincidences occur, the transmitter may either delay the new event to prevent a collision or dump the old event to preserve timing.

Transmitting events immediately by giving cells **free access** shortens latency. As such an event-driven operation does not follow a predetermined (i.e., clock-driven) readout sequence, we must transmit information that uniquely identifies the event's location. These addressed events (abbreviated to address-events) can be created simply by wiring the event-generators' outputs directly to the address encoder [12]. However, when events coincide, the encoder ORs their addresses together. These collisions increase exponentially as activity increases, with the fraction of events that get through unscathed maxing out at 18% when only 50% of the transmission slots are full [26].

Preventing collisions by **arbitered access** increases throughput. An arbiter grants only one request at a time, and the encoder outputs that cell's address. On average, the wait equals the mean interval between empty transmission slots [26]. When only 5% of the slots are empty, for example, the wait is 20 slots long. For 10 000 cells, each transmission slot must be shorter than 0.01% of the average inter-event interval to handle that many cells. Hence, a 20-slot wait corresponds to just 0.2% of the population's average inter-event interval. Therefore, arbitration can potentially achieve a five-fold increase in throughput—from 18% to 95%—with negligible timing error [15].

III. Transmitter Design

To achieve the five-fold increase in throughput arbitration promises, we must ensure that it does not increase the transmission-cycle time. The first implementation of a 2-D arbitered address-event transmitter, by Mahowald and Sivilotti, yielded a disappointingly long cycle time of 2 μs [6]. Because, for an array with n cells, a 1-in-$\sqrt{\mathsf{n}}$ arbiter was first used to pick a row and then a second 1-in-$\sqrt{\mathsf{n}}$ arbiter was used to pick a cell in that row. As a 1-in-m arbiter is built from $\mathsf{m}-1$ 1-in-2 arbiters, organized in a binary tree, this hierarchical scheme cuts the number of 1-in-2 arbiters from $\mathsf{n}-1$ to $2(\sqrt{\mathsf{n}}-1)$. Unfortunately, the cycle time suffered as all $\log_2(\mathsf{n})/2$ levels in each arbiter tree were spanned for every event transmitted.

In previous work, we cut the arbitered address-event transmitter's transmission cycle-time from 2 μs to 730 ns in the same 2-μm technology by exploiting locality [27]. That is, arbitrating at the lowest level of the arbiter tree for inputs next to each other, at the second level for inputs two to three places apart, and so on—only two levels are spanned on an average. We went on to reduce the cycle time to 420 ns by exploiting locality inside the array as well [27]. That is, servicing all active cells in a selected row before redoing the row arbitration.

Our present goal is to further optimize the arbitered address-event transmitter's row–column architecture. Having exploited locality in the arbiter and the array, transmission speed is now primarily limited by the rate at which events are read out of the array. Here, we break this array-cycling limit, realizing three enhancements in all, as follows:

1) reading a row's events in parallel boosts capacity by $\sqrt{\mathsf{n}}$;
2) bundling them into a single row-wide word eliminates the column-select lines;
3) multiplexing row and column addresses cuts output pads in half.

We alluded to Optimizations 1 and 3 in Section I. Optimization 2 is a direct consequence of Optimization 1—selecting an entire row instead of a single cell.

A preview of the transmitter architecture we developed is shown in Fig. 1. In this section, we derive programs that describe the behavior of each of these blocks by following a synthesis methodology for asynchronous digital VLSI systems developed by Martin [28] (tutorial examples are provided in [15]). His methodology involves applying a series of program decompositions and transformations, starting from a high-level specification. As each step preserves the logic of the original program, the resulting circuit is correct by induction. Thus, it is unnecessary to deduce how these hardware processes behave when executed in parallel, which is extremely difficult. After decomposing the transmitter specification into a set of concurrent one-line programs, we transform these one-liners into hardware processes in Section IV.

A. High-Level Specification

We start by writing a high-level specification in the concurrent hardware processes (CHP) language, a hardware description language for asynchronous systems [28]. In CHP, logic circuits "execute" concurrent programs. For example

$$\mathsf{RLY}(\mathsf{n}) \equiv *[A?x; B!x; C; \ldots].$$

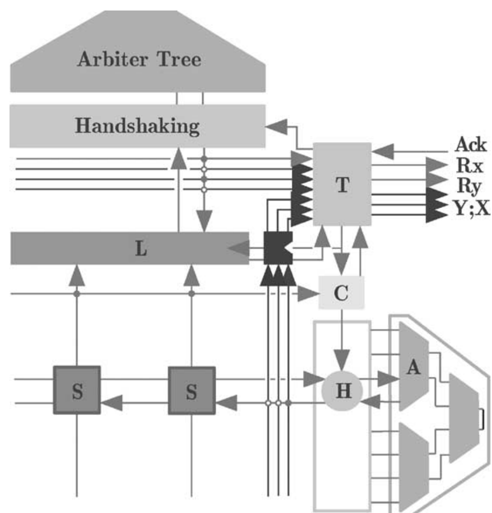


Fig. 1. Transmitter architecture: a interface circuit (H) relays requests to the row arbiter (A) and permits that row to output its address and its events (S) when the arbiter acknowledges. The events' column addresses are generated by the same procedure, after latching the row's state (L). A two-way multiplexer (T) outputs row (Y) and column (X) addresses, using separate requests lines (Ry,Rx); they share a single acknowledge line (Ack). Meanwhile, a controller (C) cycles the array to another row.

The program, or **process**, is named RLY and its argument is named n; process and argument names are always set in upper and lower case sans-serif font, respectively. As we are describing hardware here, you should think of RLY(n) as a call to a silicon compiler that lays out a circuit with, for instance, an n-bit wide datapath. $*[\cdots]$ denotes infinite **repetition**; this demarcates the body of the program. Semicolons (;) denote **sequential** execution. $A?x$ **inputs** data from a **port** named A and stores it in a local **variable** named x; port and variable names are always set in italicized upper and lower case roman font, respectively. Similarly, $B!x$ **outputs** the data stored in x on port B. C is a **dataless** communication on port C; its only effect is to synchronize the two processes whose ports are connected together. That is, this process waits until the other one gets to the corresponding point in its program, or vise versa. In the text, we will write "port C" to distinguish the port itself from a communication performed on that port, which we write simply as "C." There is no such ambiguity in the code, as only communications can appear in the body of the program.

A high-level block diagram of the address-event transmitter is shown in Fig. 2. We use **arbitration** $[G_1 \rightarrow S_1 | G_2 \rightarrow S_2 | \cdots | G_{\mathsf{n}} \rightarrow S_{\mathsf{n}}]$ to choose an active cell. It picks a guard G_j that is true and executes the corresponding program segment S_j.[1] In this case, the guard is the **probe** $\overline{P_j}$ which evaluates to true when there is a communication pending on port P_j (i.e., the other process is waiting). Also, the program segment communicates on that port and outputs its address simultaneously; **parallel** lines ($\|$) are used to denote this. Thus, we have

$$\mathsf{AEXT(n)} \equiv *\big[\big[\overline{P_1} \rightarrow A!\mathsf{enc}(1) \| P_1 | \overline{P_2} \rightarrow A!\mathsf{enc}(2) \| P_2 \cdots | \overline{P_{\mathsf{n}}} \rightarrow A!\mathsf{enc(n)} \| P_{\mathsf{n}}\big]\big] .$$

The address is returned by a function call $(\mathsf{enc}(\cdot))$ that converts a one-hot code (n-bit) to a binary one ($\log_2(\mathsf{n})$-bit).

[1]If all the guards are false, it waits for at least one to become true.

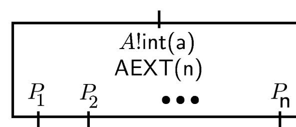


Fig. 2. Transmitter specification: when a communication occurs on dataless port P_j, its address j is output on port A as an a-bit word.

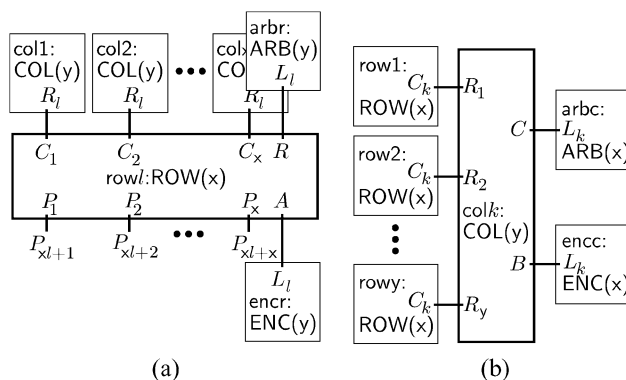


Fig. 3. Row-column organization. (a) The lth row services event generators $\mathsf{x}l + 1$ to $\mathsf{x}(l + 1)$ through its P_k-ports and communicates with x columns through its C_k ports. It also communicates with the row arbiter (arbr) and the row encoder (encr). (b) The kth column communicates with y rows through its R_l ports and communicates with the column arbiter (arbc) and the column encoder (encc) as well.

Alternatively, the transmitter process may be described succinctly using the CHP **replication** construct: $\langle \diamond k : 1..\mathsf{n} : S_k \rangle \equiv S_1 \diamond S_2 \diamond \cdots \diamond S_{\mathsf{n}}$, where each S_j is a program segment and $\diamond$ is any operator that can be concatenated. As the arbitration operator (|) can be concatenated, we have

$$\mathsf{AEXT(n)} \equiv *\big[\big[\big\langle | j : 1..\mathsf{n} : \overline{P_j} \rightarrow A!\mathsf{enc}(j) \| P_j \big\rangle\big]\big] .$$

The next step in the synthesis procedure is to decompose this high-level specification into a hierarchy of concurrent processes. These processes' ports are then connected together by **channels**. We present this connectivity information pictorially. These figures also give the names of **instances** (e.g., aer : AEXT(n) specifies an instance of AEXT(n) named aer) and their ports' **data types** (e.g., $A!\mathsf{int}(8)$ specifies that port A outputs bytes). Ports that are not defined as either input or output are dataless—by default. Port names that appear inside a box are local to that instance; those outside are local to the process within which that instance occurs.

B. Reorganizing Into Rows and Columns

Here, we decompose AEXT(n) into separate row, column, arbiter, and encoder processes, named ROW(x), COL(y), ARB(m), and ENC(m), respectively. These processes are connected as shown in Fig. 3. This decomposition is accomplished through four program transformations. For the first transformation, we reorganize AEXT(n)'s dataless ports into y rows and x columns. With this y × x array, we have to use a 1-in-y arbiter to choose a row and then use a 1-in-x arbiter to choose one of the ports in that row. Hence, we must **or** (denoted by $\vee$) together all requests within each row to generate requests for the row arbiter. We also need to provide separate outputs,

ports A_1 and A_2, for row and column addresses, respectively. Thus, we have

$$\mathsf{AEXT}(\mathsf{y},\mathsf{x}) \equiv *[[\langle|l:1..\mathsf{y}:\langle\vee k:1..\mathsf{x}:\overline{P_{l\cdot k}}\rangle \to A_1!\mathsf{enc}(l) \\ \| [\langle|k:1..\mathsf{x}:\overline{P_{l\cdot k}} \to A_2!\mathsf{enc}(k)\| P_{l\cdot k}\rangle]\rangle]]$$

where $\mathsf{y}\times\mathsf{x}=\mathsf{n}$ and $l\cdot k=\mathsf{x}l+k$.

For the second transformation, we implement arbitration in a separate process

$$\mathsf{ARB}(\mathsf{m}) \equiv *[[\langle|j:1..\mathsf{m}:\overline{L_j} \to L_j; L_j\rangle]] .$$

It performs the second communication to ensure that the dataless port it picked has been completely serviced before it picks another. Two instances of $\mathsf{ARB}(\mathsf{m})$, with $\mathsf{m}=\mathsf{y}$ or x, are used for row and column arbitration, respectively. To communicate with these processes, we provide the remaining array process, called $\mathsf{ARY}(\mathsf{y},\mathsf{x})$, with y dataless ports (R_l) and x dataless ports (C_k), respectively.

Once the row arbiter picks a row, we can use **concurrency**, $[G_1 \to S_1, G_2 \to S_2, \cdots, G_\mathsf{n} \to S_\mathsf{n}]$, to service every port in that row that has a communication pending, before picking another row. This construct executes, concurrently, all program segments, S_j, whose guards, G_j, are true.[2] Thus, we have

$$\mathsf{ARY}(\mathsf{y},\mathsf{x}) \\ \equiv *[[\langle,l:1..\mathsf{y}:\langle\vee k:1..\mathsf{x}:\overline{P_{l\cdot k}}\rangle \to R_l; A_1!\mathsf{enc}(l) \\ \| [\langle,k:1..\mathsf{x}:\overline{P_{l\cdot k}} \to C_k; A_2!\mathsf{enc}(k)\|P_{l\cdot k};C_k\rangle];R_l\rangle]] .$$

Note that a row address is output only when a new row is selected.

For the third transformation, we implement address encoding in a separate process

$$\mathsf{ENC}(\mathsf{m}) \equiv *[[\langle[\!]j:1..\mathsf{m}:\overline{L_j} \to A!\mathsf{enc}(j)\|L_j\rangle]] .$$

This process chooses one of it dataless L_j ports using **selection**, $[G_1 \to S_1 [\!] G_2 \to S_2 [\!] \cdots [\!] G_\mathsf{n} \to S_\mathsf{n}]$. Selection requires that only one guard is true at a time, which is indeed the case here, as arbitration guarantees mutual exclusion. Two instances of $\mathsf{ENC}(\mathsf{m})$, with $\mathsf{m}=\mathsf{y}$ or x, are used for row and column encoding, respectively. We provide $\mathsf{ARY}(\mathsf{y},\mathsf{x})$ with y dataless ports (A_l) and x dataless ports (D_k) to communicate with these processes. Communications on A_l and D_k, respectively, now replace the $A_1!\mathsf{enc}(l)$ and $A_2!\mathsf{enc}(k)$ communications in its program above.

For the fourth and final transformation, we break up $\mathsf{ARY}(\mathsf{y},\mathsf{x})$ into y row- and x column-processes (see Fig. 3). Dividing $\mathsf{ARY}(\mathsf{y},\mathsf{x})$'s program into subroutines yields the following code for its row and column processes

$$\mathsf{ROW}(\mathsf{x}) \equiv *[[\langle\vee k:1..\mathsf{x}:\overline{P_k}\rangle \to R; \\ A\| [\langle,k:1..\mathsf{x}:\overline{P_k} \to C_k\|P_k\rangle];R]] \\ \mathsf{COL}(\mathsf{y}) \equiv *[[\langle[\!]l:1..\mathsf{y}:\overline{R_l} \to C; D\|R_l;C\rangle]]$$

where A and D have replaced $A_1!\mathsf{enc}(l)$ and $A_2!\mathsf{enc}(k)$, respectively, instead of A_l and D_k. We parallelize this serial-array-readout design, which was implemented in [27], in the next subsection.

[2] This construct is not supported by CHP. Its use is discouraged because the negated probes used to determine ineligibility can change from true to false at any time. Concurrency waits for at least one guard to become true if necessary, just like arbitration does.

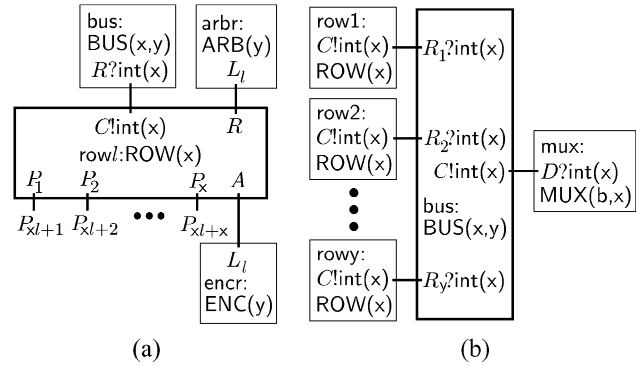


Fig. 4. Parallel readout. (a) ROW's C_k ports have been combined into a single port (C) that outputs an x-bit word. (b) Column processes are replaced by a bus (bus) that transfers x-bit words to a mux (mux).

C. Reading the Array in Parallel

Here, we transform $\mathsf{ROW}(\mathsf{x})$ and $\mathsf{COL}(\mathsf{y})$ to read all of a selected row's dataless ports in parallel, an innovation introduced in this work (initially reported in [22]). This parallel read is accomplished by merging the x instances of $\mathsf{COL}(\mathsf{y})$ into a single x-bit-wide bus, called $\mathsf{BUS}(\mathsf{y},\mathsf{x})$, and modifying $\mathsf{ROW}(\mathsf{x})$ accordingly, as shown in Fig. 4. An x-bit integer, named w, can represent the state of a row's x dataless ports (P_k). We denote w's kth bit by $w.k$. If we **set** this bit $(w.k\uparrow)$ when the probe, $\overline{P_k}$, becomes true, it may serve as a proxy for $\overline{P_k}$ in the previous $\mathsf{ROW}(\mathsf{x})$ program. We just have to remember to **clear** all the bits ($w.k\downarrow$ or $w:=0$) once the row is read. Thus, we have

$$\mathsf{ROW}(\mathsf{x}) \equiv *[[\langle,k:1..\mathsf{x}:\overline{P_k} \to w.k\uparrow\rangle];R; \\ A\|C!w\|[\langle,k:1..\mathsf{x}:w.k \to P_k\rangle];R\|w:=0] \\ \mathsf{BUS}(\mathsf{y},\mathsf{x}) \equiv *[[\langle[\!]l:1..\mathsf{y}:\overline{R_l} \to C!(R_l?)\rangle]] .$$

One instance of $\mathsf{BUS}(\mathsf{y},\mathsf{x})$ replaces the x instances of $\mathsf{COL}(\mathsf{y})$ we had before.

We transfer the row word w to a latch whose cells execute the code given for $\mathsf{COL}(\mathsf{y})$ previously, with each bit, $w.k$, serving as a proxy for $\overline{R_l}$. Hence, we test $w.k$ instead of probing the kth column's R_l ports and clear $w.k$ instead of doing the R_l communication. Thus, we have

$$\mathsf{LTH}(\mathsf{x}) \equiv *[R?w; [\langle,k:1..\mathsf{x}:w.k \to C_k; D_k\|w.k\downarrow;C_k\rangle];R] .$$

The second R communication now signals that the latch is empty. $\mathsf{LTH}(\mathsf{x})$ is connected as shown in Fig. 5(a). It receives the row's data from $\mathsf{MUX}(\mathsf{b},\mathsf{x})$, a process that we shall describe shortly. This completes our transformation of $\mathsf{ROW}(\mathsf{x})$ and $\mathsf{COL}(\mathsf{y})$ to support parallel reading.

Our final transformation is to transmit row and column addresses sequentially, over the same output; this innovation is revealed here for the first time. The addresses are multiplexed by a separate process called $\mathsf{MUX}(\mathsf{b},\mathsf{x})$, where $\mathsf{b}=\max(\log(\mathsf{x}),\log(\mathsf{y}))$ specifies the number of address bits. As shown in Fig. 5(b), while it relays addresses from the row encoder (port R) or the column encoder (port C) to the transmitter's output (port T), $\mathsf{MUX}(\mathsf{b},\mathsf{x})$ also relays row-data from $\mathsf{BUS}(\mathsf{y},\mathsf{x})$ (port D) to $\mathsf{LTH}(\mathsf{x})$ (port S) as well.

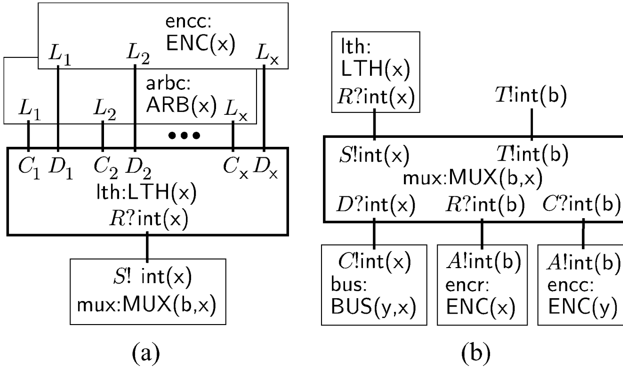


Fig. 5. Row-word latch and burst generator: (a) A latch (lth) receives row-words from the mux and communicates with the column arbiter (arbc) and the column encoder (encc). (b) A mux (mux) receives row and column addresses from the encoders (encr and encc) and outputs them on the transmitter's T port; it also relays row-words from the bus (bus) to the latch (lth).

Coordinating these two procedures makes it possible to send the empty signal, a reserved address named ϕ, when LTH(x) becomes empty. Thus, we have

$$\mathsf{MUX(b,x)} \equiv *[S!(D?) \| T!(R?); S; T!\phi] \, \| \, *[T!(C?)] \, .$$

Recall that the second S communication, which matches LTH(x)'s second R communication above, signals that the latch is empty. Column addresses are handled by a concurrent process, under the assumption that they show up after the row address does. This timing assumption can be avoided by forcing MUX(b, x) to wait for the new row address to appear, ignoring all column addresses after it relays ϕ.

In summary, we have decomposed AEXT(n) into six concurrent processes: y instances of ROW(x), one of BUS(y, x), one of MUX(b, x), and one of LTH(x). The remaining two are ARB(m) and ENC(m); there are two instances of each, with m equal to x or y. The next step in the synthesis procedure is to compile these CHP programs into hardware.

IV. Transmitter Implementation

Electrically, processes **set** (bo+) or **clear** (bo−) an output signal (bo), or wait for an input signal (bi) to become **true** ([bi]) or **false** ([˜bi]); tilde (˜) denotes logical **complement**. To communicate, they must perform complementary **four-phase** sequences of actions and waits: *[ao+; [ai]; ao−; [˜ai]] on an **active** port A and *[[pi]; po+; [˜pi]; po−] for its **passive** counterpart P, where *[...] denotes repetition, just like in CHP. We always append i and o to the port's name to indicate its input and output signals, respectively. Such signal names are always set in lower case typewriter font. The active port's output signal (ao) is commonly called **Request**; the passive port's (po) is the so-called **Acknowledge**. At the signal-level, we refer to A as (ao, ai) and to P as (pi, po)—request first and acknowledge second in both cases.

We have three choices of signal representations for data. (1) **Bundled-data** requires a single line per bit, in addition to the request and acknowledge signals. The data is valid when the request signal is set; otherwise it is invalid. (2) **Straight-data** dispenses with the request signal. Instead, all zeroes signifies invalid data; any other word is considered valid. Both representations require matched delays—for data as well as request. (3) **Dual-rail** achieves delay-insensitive operation by encoding each bit using two lines: bit-is-true and bit-is-false (denoted by appending t or f). The data are invalid when both are cleared; setting either transmits a one or a zero.

Handshaking expansion (HSE) is the procedure whereby each communication in our CHP programs is fleshed out into a full four-phase request–acknowledge sequence. Following Martin's synthesis procedure [28], we make two choices when we perform HSE. First, we make output ports active and input ports passive.[3] The only exception is a port that is probed must be passive, as the **probe** is implemented simply as [pi]. Symmetric links—dataless ports that are not probed on either end—are dealt with on a case by case basis. Second, we use the second half of the four-phase handshake to implement a second communication on the same port—a **two-phase** handshake—if these communications always occur in pairs. This optimization is possible because the second half just returns the signals to their initial state. So we are free to clear them whenever it is convenient to do so, a process known as **reshuffling**.

The final step in Martin's synthesis procedure is compiling HSE sequences into production-rule sets (PRS), which are straightforward to implement with CMOS transistors. A **production-rule**, e → b−, clears a bit, b, when a boolean expression, e, becomes true. We write ˜e → b+ to set the bit when the expression is false. A nFET implements the former rule while a pFET implements the latter—the two rules together correspond to an inverter. Logical **and** and **or** (denoted by & and |, respectively, in PRS, or HSE) are implemented by connecting FETs in series and in parallel. If both pull-up and pull-down chains may both be inactive at the same time, a weak feedback-inverter must be added to overcome their leakage currents. Such outputs are said to be **state holding**, as opposed to **combinational**; the feedback inverter is called a **staticizer**. **Active low** signals are allowed in PRS and at the circuit level; their names have an underscore prepended (e.g., _ai).

We present only the final HSE sequences and the synthesized circuits in this section. Details of how we arrived at these reshufflings and how we compiled them into PRS are in the Appendix. We recommend that you refer to Fig. 6 to see how these circuits interact as you read their descriptions. To facilitate this, we include the block labels in this figure in HSE sequences and in subsequent figure captions.

A. Reading

The CHP program for ROW(x) [see Section III-C and Fig. 4(a)] calls for us to set a bit $(w.k)$ when the event-generator initiates a communication (on port P_k) and readout this data onto the column lines (port C) when this particular row is selected by the arbiter (port R). ROW(x) must communicate with the address encoder as well (port A). These operations are implemented in this section, together with the CHP for BUS(y, x).

We made port P_k passive and ports R, C, and A active and we chose a straight-data representation for port C, where all zeroes indicates invalid data. We separated event-generator and

[3]Our choice is arbitrary—the direction that data flows is not necessarily restrictive.

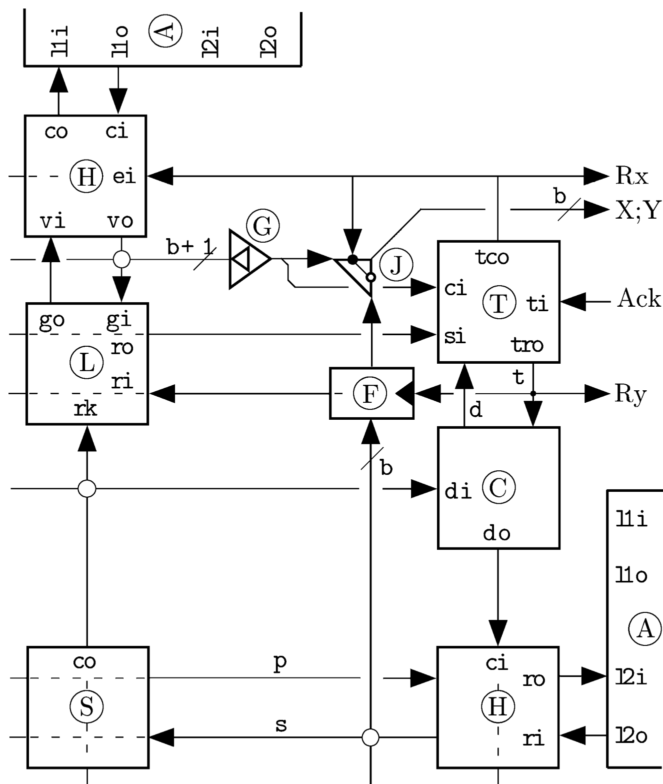


Fig. 6. Transmitter schematic. ROW consists of event-generator interfaces (S) and an arbiter interface (H). ARB consists of 1-in-2 arbiter cells (A). LTH's bit cells consist of a memory (L) and an arbiter interface (H). BUS consists of a set of wires (from S to L). ENC (two instances) consists of b address lines (extra column-address line is request) and combinational logic (represented by discs). Latch (F) stores the row address while staticizers (G) hold the column address. MUX consists of two controllers: one (T) switches the address-mux (J), the another (C) cycles the array.

event-arbiter signals into two sequences, introducing two intermediaries, p and s (see Fig. 6), for them to communicate with: p goes high when an event occurs anywhere in this row while s goes high when the row is selected (see Appendix, part A). This partitioning resulted in the following reshuffled HSE sequences

```
# row (S,H) #
  *[[pi];w+;[s];co+,po+;[~pi];w-;[~s];co-,po-]
  ||*[[p];ro+;[ri&~ci];s+;[ci&~p];ro-;[~ri];s-]
```

where || denotes parallel execution, just like in CHP. p is the OR of all the w bits. For brevity, the k subscript has been suppressed and A has been omitted: ao transitions at the same time s does and ai is combined with ci (see Appendix, part A).

Compiling the first sequence into PRS (see Appendix, part A) yielded the circuit shown in Fig. 7(a). These two gates are asymmetric variations of the **C-element**, whose output is set when both inputs are high and cleared when both are low (i.e., {a&b → c+,~a&~b → c−}); they are called aC-elements. Initially, s is low, so when _pi becomes low, w goes high, which prompts s to go high. Consequently, co and po go high, which prompts _pi to go high. As a result, w is cleared, which prompts s to go back low, thereby clearing co and po to terminate the cycle.

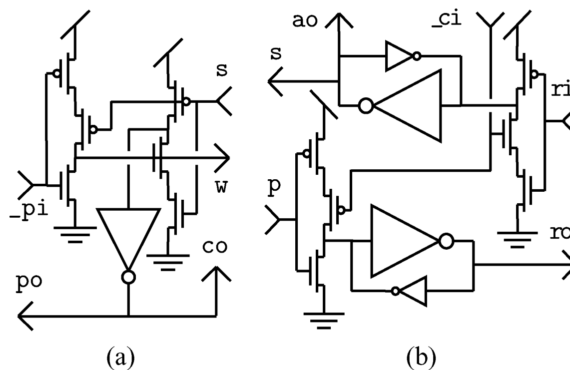


Fig. 7. Event generator [S] and arbiter [H] interfaces. (a) Interfaces event-generator (_pi,po) with arbiter-interface (w,s) and latch-cell (co). (b) Interfaces row (p,s), or latch-cell, with arbiter (ro,ri), address encoder (ao), and mux (_ci).

We eliminated the staticizers to save space, but this simplification produced a race condition when the first gate is disabled by s+ [see Fig. 7(a)]. If _pi has not discharged all the way to ground, the pull-down continues to pass current and clears w. Thus, it is important for the event-generator to produce a fast, clean, downward transition [8], [25] at _pi. If not, the staticizers must be included. The absence of staticizers also makes the circuit susceptible to charge sharing, which can be largely avoided by placing the series-connected n and p transistors in the order shown.

Compiling the second sequence into PRS (see Appendix, part A) yielded the circuit shown in Fig. 7(b), which also consists of two aC-elements. When p becomes high, ro goes high, which prompts ri to go high. As _ci is high initially, s and ao go high, which prompts p and _ci to go low. Thus, ro goes low, causing ri to go back low, which clears s and ao. A new cycle can now begin, but s cannot go high until _ci goes back high.

We relay requests from the row's multiple event-generator interfaces to this arbiter-interface using the circuit shown in Fig. 8. This circuit ORs together all w bits in that row (they are tied to l1i, etc.) to generate p (tied to ro) and broadcasts the s signal (tied to ri).[4] This staticized design is more power-efficient and noise-immune than the nMOS-style wired-OR used previously [6], [10], [15]. The address encoder is implemented as described in [15].

BUS(y, x) merges words form the y ROWs onto its C port, which we chose to be active [see Section III-C and Fig. 4(b)]. We implemented this merge by feeding ROW's straight-data outputs to x staticized wired-OR gates, with y inputs each (see Fig. 8), connected in a column-wise fashion. Instead of steering the acknowledge signal to the row that was read, it is tied directly to all the rows' arbiter-interfaces (see Fig. 6). While simplifying the steering circuit to a single wire, this global acknowledge signal also blocks newly selected rows from proceeding until on-going column communication is completed, making the aggressive arbiter-interface reshuffling presented above safe (see Appendix, part A).

[4]The select signal is not restricted to the active cells because it is used to prevent inactive ones from becoming active. Thus, this broadcast deals with the negated probe instability that plagues concurrency.

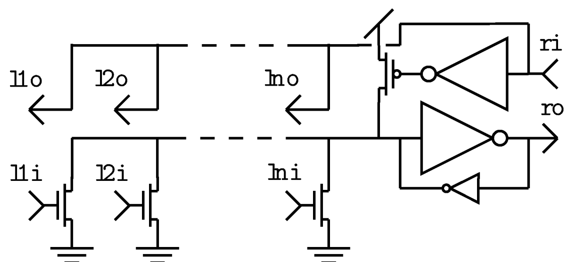


Fig. 8. ORing request signals. ORs multiple requests, l1i, l2i, ..., or lni, together to create a single request, ro, and broadcasts the acknowledge, ri, to all n ports. ri clears ro—but not until all lki are cleared. Because, the pFET is not strong enough to overcome an nFET.

B. Choosing

ARB(m) is built out of m − 1 ARB(2) cells, as shown in Fig. 1 (see [15] for m not-a-power of two). The 1-in-2 arbiter cell is described by

$$\mathsf{ARB}(2) \equiv *[[\overline{L_1} \to R; L_1; L_1; R | \overline{L_2} \to R; L_2; L_2; R]].$$

Ports L_1 and L_2 are connected to its daughters' R ports while port R is connected to its parent's L_1 or L_2 port. Only after communicating on R does it perform a communication pending on L_1 or L_2, arbitrating between them if necessary. Thus, requests are relayed up the tree by probing the R-to-L channels, while choices are steered down the tree by communicating on the same channels. The second pair of L and R communications guarantees mutual exclusion.

We decomposed ARB(2) into two processes by isolating its L_1-to-R and L_2-to-R communications, and provided a third process to arbitrate between these two

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

Ports in different processes with the same name are connected together. For the communication processes, we made ports L_1 and L_2 passive and ports A_1, A_2, and R active. After reshuffling to optimize performance (see Appendix, part B), we obtained the following HSE sequence for the first communication process:

```
# arb (A) #
*[[l1i&~ri];a1o+;ro+;[ri&a1i];l1o+;
[~l1i];a1o-;ro-;[~a1i];l1o-].
```

The second communication process is identical; just replace 1 with 2. Their two ro signals are ORed together to generate a single request signal.

Compiling the sequence above into PRS (see Appendix, part B) yielded the circuit shown in Fig. 9. Two cross-coupled NAND gates perform arbitration [16]. Their inputs are active-high and their outputs are active-low—complementary to a set–reset flip-flop. The aC-elements activate these inputs when requests are received, provided the parent's acknowledge (_ri) is inactive (i.e., high). The NAND gates' outputs drive a circuit that steers the parent's acknowledge (_ri) to either daughter (l1o or l2o). This steering circuit NORs these active-low signals,

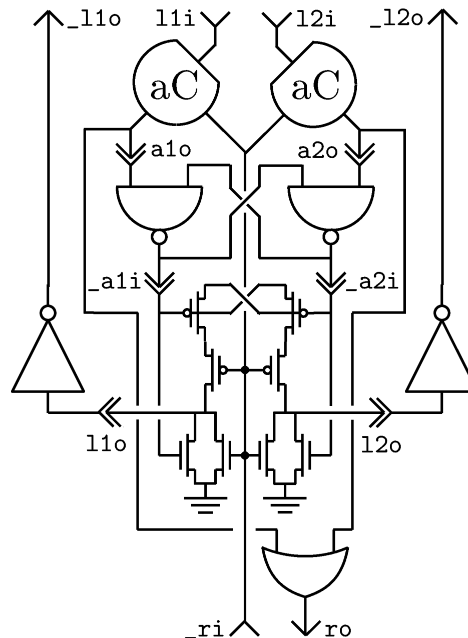


Fig. 9. Two-input arbiter cell [A]: interfaces two daughter cells, (l1i,_l1o) and (l2i,_l2o), with a parent cell, (ro,_ri), using two asymmetric C-elements (aC), a pair of cross-coupled NAND gates, a steering circuit, and an OR gate.

_a1i and _a2i, with _ri to produce the outgoing acknowledges, l1o or l2o. To filter out metastability, the NOR-gates' pull-ups are not powered up unless _a1i and _a2i differ by more than the threshold voltage [16], [28].

When these 1-in-2 arbiter cells are connected in a binary tree, requests are selected by a post-order traversal. That is, a node is visited, and then, its daughters are visited, and so on, recursively. However, a daughter that is not requesting is not visited and a daughter that makes another request is not revisited until the entire tree has been traversed. Each daughter is visited only once because the [l1i&~ri] wait in #arb (A)# above blocks a second request from being serviced with the same ri acknowledge signal, unlike the greedy arbiter design presented in [15], [27], which would revisit the same daughter over and over again. Complete traversal makes this new arbiter design a fair one, in that it will not service the same client again until all those waiting have been serviced. Fairness is critical if parallel readout is to be fully exploited, as discussed in the companion paper [24].

This fair arbiter design is optimized for speed, in that new requests can propagate up the tree while old ones are still being serviced. It does not require requests at all levels of the tree to be cleared before the acknowledges are cleared (see Fig. 1). Traversing the entire tree like this—starting at the bottom and propagating all the way up to clear the requests and then starting at the top and propagating all the way down to clear the acknowledges—would be painfully slow. Instead of waiting for its parent's acknowledge to clear before it clears its own acknowledge, the cell clears its acknowledge as soon as its daughter's request clears (see #arb (A)#). However, it blocks new requests until its parent's acknowledge clears, which just requires the cell to clear its own request. The cell does this once both of its daughters' requests are cleared. Thus, new requests propagate up until they encounter a cell whose other request is currently selected.

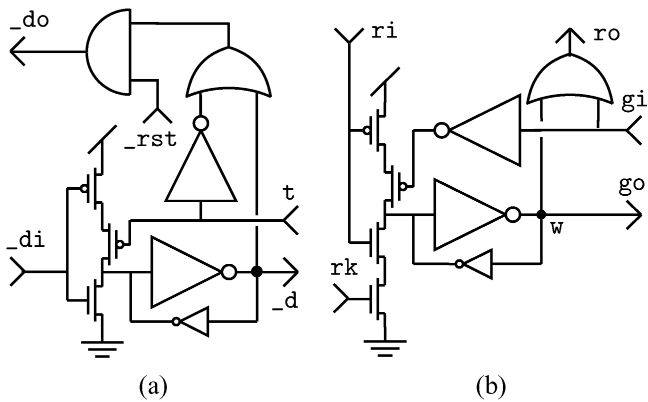


Fig. 10. Data-transfer [C] and latch [L] circuits. (a) Interfaces array (_di,_do) with mux (_d,t). _do is forced low during reset. (b) Stores (w) row cell's output (rx) under control of mux (ri,ro) and communicates with column arbiter interface (go,gi).

In fact, they can get all the way up to the top cell, even while it is servicing the other half of the tree.

C. Writing

The CHP program for LTH(x) calls for us to store data read from a row (port R) and then communicate with the arbiter (port C_k) and the column encoder (port D_k). These communications are performed for each bit $(w.k)$ that is set, and then the bit is cleared. When all the bits have been cleared, a second R communication is performed to signal that the latch is empty. These operations are implemented in this section, together with the part of MUX(b, x) that coordinates them.

MUX(b, x) hands data from BUS(y, x) to LTH(x) $(S!(D?))$, converting it from a straight-data representation to a bundled-data one (see Section III-C and Fig. 5(b)). We separated the read operation on MUX's passive D port and the write operation on its active S port into two HSE sequences. We also introduced a pair of local variables d and t for them to communicate with (see Fig. 6). The read sequence takes d high when the data appears; the write sequence responds by taking t high when the data is latched. We present the read sequence below but we postpone implementing the write sequence till Section IV-D, in order to synchronize it with row address transmission (see Section III-C). The straight-to-bundled-data converter is implemented simply by performing a bit-wise OR on the column data to generate a request signal (di below).

We obtained the following reshuffling for the read sequence (see Appendix, part D):

```
# xfr (C) #
*[[di];d+;[t];do+;[~di];d-;do-;[~t]].
```

Compiling this sequence into PRS (see Appendix, part D) yielded the circuit shown in Fig. 10(a). Initially t is low, so when _di becomes low, _d goes low, which prompts t to become high. Now, both of the OR gate's inputs are low, so it drives _do low, which prompts _di to go high, and hence _d and _do go back up. Once t goes low, a new cycle may begin. However, new column data can show up immediately after _do goes high, so this reshuffling allows us to cycle to the next row and present its data even before the latch becomes empty.

Now, we proceed with implementing LTH(x) [see Section III-C and Fig. 5(a)]. First, we decompose its CHP program into two processes

$$\mathsf{LTH(x)} \equiv \ *[R?w;[\langle, k:1..\mathsf{x}: w.k \to G_k; w.k\downarrow\rangle];R] \\ \| \langle \| k:1..\mathsf{x}: *[\overline{V_k} \to C_k; D_k \| V_k; C_k]\rangle .$$

The G_k port of the first process, which stores the bit, is connected to the V_k port of the second one, which interfaces with the arbiter. The arbiter interface turned out to be the same as that used by the rows [see Section IV-A and Fig. 7(b)]. We simply make the connections: vo = p, vi = s, co = ro, ci = ri, do = s, di = ci. For the memory cell, we used a bundled-data representation for port R and made it passive, and we made port G_k active. These choices yielded the following reshuffled HSE sequence:

```
# latch (L) #
*[[ri&rk];w+;go+;ro+;[~ri&gi];w-;go-;[~gi];
  ro-].
```

Compiling the sequence above into PRS (see Appendix, part C) yielded the circuit shown in Fig. 10(b). Initially ri is high, so when rk becomes high, w is set and go and ro go high, which prompts gi to go high and ri to go low. Thus, w and go are cleared, but ro stays high until gi becomes low. When this happens, ri goes high in response to ro going low, and now a new cycle can begin. We OR together the ro signals from all memory cells to generate a single write-acknowledge. Even though this OR-gate is triggered by the first bit that is set, the delay in clearing the request signal keeps the latch transparent for a while, giving tardy bits the chance to be written.

D. Bursting

The CHP program for MUX(b, x) calls for us to write row-words to the latch (port S), to multiplex row addresses (port R) and column addresses (port C) onto the transmitter's output (port T), and to send ϕ when the latch is empty [see Section III-C and Fig. 5(b)]. These operations are implemented in this section; reading the row's data $(D?)$ was implemented in Section IV-C.

We use the following three-wire-handshake sequence for port T:

```
  *[tro+;[ti];tro-;[~ti]]
||*[[ti];tco+;[~ti];tco-].
```

This protocol allows us to transmit multiple column addresses by executing the second sequence as many times as desired, halfway through the first sequence. The first sequence's first half transmits the row address, while the second half terminates the burst—it transmits ϕ. Fig. 6 shows the correspondence between these signals and the transmitter's Ry, Rx, and Ack signals mentioned earlier.

First, we implement the row communications, $S!(D?)$ and $T!(R?)$. We will merge reading the row's address with reading its data, and thereby use the read sequence presented in Section IV-C for $R?$ as well as $D?$. Hence, the (d, t) signals introduced above (#xfr (C)#) will serve as proxies for (ri, ro).

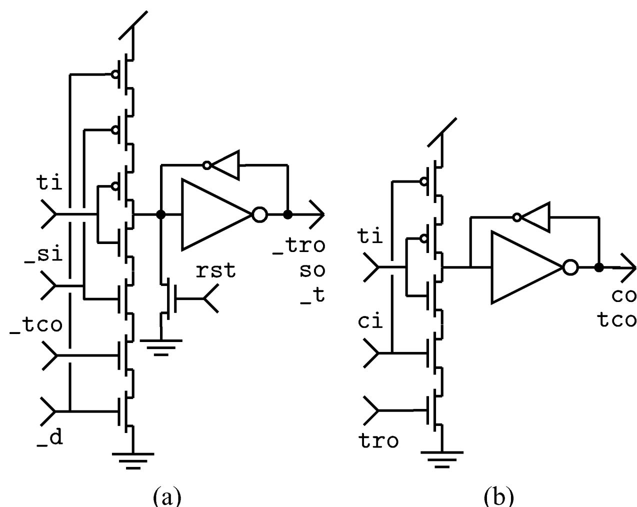


Fig. 11. Address transmission circuits [T]. (a) Synchronizes row-address transmission (_tro,ti) with latch (so,_si) and array (_d,_t). so and _t are forced high during reset. (b) Synchronizes column-address transmission (tco,ti) with column arbiter interface (ci,co) and switches the address-mux (tco).

That takes care of reading. For writing, we use the following reshuffled sequence (see Appendix, part D):

```
# yctl (T) #
*[[˜ti&d&si];tro+,so-;[ti&˜d&˜si];tro-,so+]
```

which takes care of both T and S; t transitions at the same time as tro. Compiling this sequence into PRS (see Appendix, part D) yielded the circuit shown in Fig. 11(a). Initially, so, _tro, and _t, are all high. When the row's address and data appear, _d goes low, and when they are latched, _si goes low. At this point, assuming ti is low, so, _tro, and _t go low. These signals are cleared when _si, ti, and _d, as well as _tco, go high.

Now we implement the column communication, $T!(C?)$, using the following reshuffled sequence (see Appendix, part D):

```
# xctl (T) #
*[[ti&ci];co+,tco+;[˜ci&˜ti];co-,tco-].
```

Compiling this sequence (see Appendix, part D) yielded the circuit shown in Fig. 11(b). As tro is high for the burst duration, and ti is high initially, we only have to wait for ci to become high. When that happens, co and tco go high, which switches the mux to the column address (see Fig. 6). When both ti and ci become low, these signals are cleared. And when ti and ci go high in response, a new cycle can begin. Since the mux switches back to the row address when tco is low, this address can be reread at any time. If it so desires, the receiver can buy time to do this by taking its acknowledge high a little latter.

We developed a reset strategy to recover when a row is selected but no data is delivered to the latch. This situation could arise if the event-generator's slew-rate is too slow (see Section IV-A). We activate the data-transfer circuit's array-acknowledge [_do in Fig. 10(a)] to complete the stalled cycle. We also activate the row-address transmission circuit's request [so in Fig. 11(a)] to make the latch transparent, so that the next row can be written. Since the write failed, we do not bother to clear the contents of the latch. If the latch is not actually empty (e.g., during power-up), those bits will corrupt the next burst. However, after that one is sent, everything will be fine.

V. Summary and Conclusion

We have described an address-event transmitter that reads all active cells in a selected row in parallel. Row activity is transmitted in a burst: the row address followed by a column address for each active cell, plus a termination signal. The array is cycled to the next row while these events are being transmitted, so the next burst can start as soon as this one ends. Bursts are communicated using a three-wire protocol: a row-request, a column-request, and a common acknowledge. In return for the extra request line, output pads are cut by 50%—without sacrificing throughput—as the row address is not repeated.

In terms of cell area, the cost of parallel-readout is minimal. Whereas previous transmitter designs add four transistors to the cell (reviewed in [15]), our design requires nine transistors. Both of these counts include the transistor that pulls down the row-request line but do not include the one that resets the event-generator. On the otherhand, our design requires just one line per column whereas previous designs require two, since they select columns individually. Trading a metal line for five transistors is highly favorable when wires are at a higher premium than transistors, which is increasingly the case. Thus, the increased throughput—and scalability—parallelism offers [24] is attained at little cost in hardware.

We also illustrated how to synthesize an asynchronous implementation starting from a high-level specification by way of a concrete example. The result was eight logic circuits that, together, can be used to implement a burst-mode, word-serial, address-event transmitter of any desired size. These circuits include an arbiter design that allows parallelism to be exploited fully by ensuring that a row is not reread until all those waiting are serviced. We have laid out a library of cells (in MOSIS DEEP_SUBM rules) for these circuits and written a silicon-compiler to tile them to fit any desired pixel- or array-size. Thus far, this tool has successfully compiled transmitters for three generations of chips, fabricated in 0.6-, 0.4-, and 0.25-μm technology [24].

Appendix
Logic Synthesis

When compiling HSE into PRS, we perform two passes. On the first pass, we make the wait before an action the guard for its production rule. For example, $*$[[pi]; po+; [˜pi]; po−], the passive port's sequence, is realized by the set {pi− > po+, ˜pi− > po−}, which is implemented by a wire. On the second pass, we strengthen guards that can become true at some other point in the sequence by ANDing with another boolean variable. If all signals are in exactly the same state at these two points, we add a state variable to distinguish them, setting it after we pass the first point and clearing it after we pass the second point, or vise versa.

For example, the CHP process $*[A;P]$, where A is active and P passive, as above, could be augmented with the state variable `s` as follows:

```
*[ao+;[ai];s+;ao-;[~ai];[pi];po+;[~pi];
s-;po-].
```

`s`'s state now distinguishes the point where A ends and P begins from the point where P ends and A repeats. Alternatively, the ambiguous state can be eliminated if we begin P before A ends. For example

```
*[ao+;[ai];[pi];po+;ao-;[~ai];[~pi];po-]
```

is unambiguous. This reshuffling, if acceptable, is cheaper to implement, as it does not require a state variable.

We can often avoid adding state variables by reshuffling sequences in this way, provided the change in sequencing is benign. When compiling PRS for such sequences, multiple preceeding waits are ANDed together (e.g., `ai&pi- > po+`) and preceeding actions become guards too (e.g., `po- > ao-`). Another goal of reshuffling is symmetry—clearing signals in the same order that you set them. Such symmetry makes the signals that appear in the pull-up and the pull-down the same. This duplicity usually results in a simpler implementation, as the pull-up is disabled when the pull-down is active, and vise versa.

It is sometimes possible to convert a state-holding gate into a combinational one, thereby avoiding the need for a staticizer. That is, to make the gate's pull-up (`u- > b+`) and pull-down (`d- > b-`) complementary (`d = ~u`). Such conversion is typically done by ORing terms with the pull-up (`u`) and ANDing terms with the pull-down (`d`), or vise versa. For example, {`a- > c-`, `~b- > c+`} requires a staticizer, since `a = ~(~b)` is not an identity. However, {`a&b- > c-`, `~a|~b- > c+`} is combinational, since `a&b = ~(~a|~b)` is an identity. In fact, that is a NAND gate. These added terms must have a benign effect, such that `u|w = u` at all points in the sequence, where `u` is the original guard and `w` is the weakening term.

A. Row

Making ROW's P_k port passive, and its R, C, and A ports active (see Section III-C), yielded this single-bit HSE sequence

```
*[[pi];w+;ro+;[ri];co+,po+;[~pi];w-;
[ci];co-,po-;[~ci];ro-;[~ri]]
```

where A has been omitted for the time being and the k subscript is suppressed for brevity. If we move the second R communication (two-phase) ahead of the second C communication, the arbiter can start selecting the next row earlier. However, we must ensure that this newly selected row does not interrupt an on-going column communication. It can be blocked by advancing `[~ci]` forward to where `[ri]` occurs in the next cycle, which also provides more time to complete the column communication. Thus, the sequence becomes

```
*[[pi];w+;ro+;[ri&~ci];co+,po+;[~pi];w-;
[ci];ro-;[~ri];co-,po-]
```

where `ci` is broadcast to all the rows. Next, we augment the sequence with row-wide event (`p`) and selection (`s`) signals to support multiple bits. For `p`, which is the OR of all the bit-level `w` signals, we insert "`p+;[p];`" after "`w+`" and "`p-;[~p];`" after "`w-`". And for `s`, which mirrors `ri`, we insert "`s+;[s];`" after "`..~ci];`" and "`s-;[~s];`" after "`..~ri];`". Thus, we obtain

```
*[[pi];w+;p+;[p];ro+;[ri&~ci];s+;[s];co+,po+;
[~pi];w-;p-;[~p];[ci];ro-;[~ri];s-;[~s];
co-,po-].
```

Finally, moving the row-level parts (i.e., from `[p]` to `s+` and `[~p]` to `s-`) into a separate (arbiter-interface) sequence yielded the reshuffling presented in Section IV-A (`#row(S,H)#`).

We compiled our final reshufflings into the following PRS:

```
pi&~s->w+          s&w->co+,po+
   p->ro+        r&~ci->s+
  ~pi->w-            ~s->co-,po-
~p&ci->ro-          ~ri->s-.
```

We strengthened the guard of `w+` with `~s` to ensure that only those cells that were active when the row was selected participate, as required by concurrency. And we strengthened the guard for `co+`, `po+` with `w`, to ensure that only active cells respond. The circuits are shown in Fig. 7.

We include the A communication by observing that it occurs simultaneously with $C!w$ (see Section III-C). Hence, we can activate `ao`, as well as `co`, with `s`, and combine `ai` with `ci` using a C-element.[5] Alternatively, since we activate the row and the encoder at the same time, we can assume the column bus's acknowledge `ci` indicates that both the row's state and its address have been latched. This timing assumption eliminates the C-element, but requires that we compensate for worst-case timing-differences between the address-encoding and data-transfer processes.

B. Arbiter

Making ARB(2)'s L_1 and L_2 ports passive and its R port active yielded this HSE sequence for the first communication process (see Section IV-B)

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

where port A_1 (and A_2) is passive. If we execute `ro+` without waiting for `[a1i]`, we will allow the upper levels to make decisions concurrently. We can maintain mutually exclusive access to R by delaying `[~ri]` until the next cycle. This so-called **lazy-active** reshuffling (e.g., `*[[~ri];ro+;[ri];ro-]`) gave us

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

However, the other daughter is not excluded if her request (i.e., `l2i`) becomes active while `ri` is still false. In that case,

[5]A two-input gate whose output is set when both inputs are high and cleared when both are low.

we can kill two birds (service both daughters) with one stone (a single R communication). That is, once `a1o`− fires, the arbitration process will make `a2i` true, allowing the other communication process to get past [`ri&a2i`], where it is held up, and select the other daughter.

We can also get `a1o`− to happen faster by postponing `l1o`− till the end and clearing `a1o` and `ro` in the same order that we set them. These changes also make the sequence more symmetric, which simplifies the logic. Thus, we obtained the reshuffled HSE sequence presented in Section IV-B (#`arb(A)`#). We compiled that sequence into this PRS

```
   l1i&_ri->a1o+         ~l1o->_l1o+
      ~l1i->a1o-          l1o->_l1o-
~_a1i&~_ri->l1o+       a1o|a2o->ro+
  _a1i|_ri->l1o-    ~a1o&~a2o->ro-
```

where we simply OR (AND) the two `ro` pull-ups (pull-downs) together. Weakening `l1o`−'s guard with `_ri` is since `_a1i` becomes true first. Because, as you can see from the circuit (Fig. 9), `a1o`− propagates through only two gates to set `_a1i` but it propagates through one gate in this cell plus three gates at the next level—and an inverter—to set `_ri`.

C. Latch

Making LTH's R port passive, and using a bundled-data representation, yielded this HSE sequence for its memory cell

```
*[[ri&rk];w+;ro+;go+;[gi];go-;[~gi];w-;
[~ri];ro-]
```

where port G_k is active (the `k` subscript is suppressed). Moving the second two-phase G communication to the middle of the second (two-phase) R communication eliminates an ambiguous state. However, this reshuffling implies that the cell cannot start the second G communication (with the arbiter-interface) before the second R communication (with the mux) starts. The consequences of synchronizing these two-phase communications are dealt with in Appendix , part D. Swapping `ro`+ with `go`+ and `w`− with [~`ri`] reduces asymmetry; both swaps are benign. These changes yielded the reshuffling given in Section IV-C (#`latch(L)`#).

We compiled our final reshuffling into the following PRS:

```
 ri&rk->w+       w->go+       go|gi->ro+
~ri&gi->w-      ~w->go-     ~go&~gi->ro-.
```

Weakening `ro`+'s guard is safe because `gi` happens later; strengthening `ro`−'s makes the gate combinational. The circuit is shown in Fig. 10(b).

D. Mux

The CHP for MUX(b, x)'s row communications calls for parallel execution of $T!(R?)$ and $S!(D?)$ (see Section III-C). However, we allowed them to run in parallel only after the row's address and its data are latched, since we wished to merge the reads (see Appendix, part A). With ports D and R passive and ports T and S active, this strategy is realized by the HSE sequence

```
*[[di];so+;[si];(do+;[~di];do-)||
(tro+;[ti];so-;[~si];tro-;[~ti])]
```

where (`di`, `do`) serve the merged D–R port. We broke this sequence up into two concurrent read and write sequences and synchronized them with two new variables, `d` and `t`

```
  *[[di];d+;[t];do+;[~di];d-;do-;[~t]]
||*[[d];so+;[si];t+;tro+;[ti];so-;
[~d&~si];tro-;[~ti];t-]
```

`d`− could have been executed anywhere between [`t`] and [~`t`]; we went with symmetry. The first sequence is identical to the read sequence given in Section IV-C (#`xfr(C)`#).

We reshuffled the write sequence further. Moving `so`+ immediately after the previous cycle's [~`si`] makes the latch transparent as soon as it becomes empty. This move allows us to merge [`d`] and [`si`], and we can consolidate [~`ti`] as well by using the lazy-active reshuffling. Postponing [`ti`] till after `so`− allows the second (two-phase) S communication to occur as soon as we start transmitting the row's address. These optimizations yielded the write sequence given in Section IV-D (#`yctl(T)`#).

This reshuffling deals with the consequences of synchronizing LTH's G and R communications. That is, we do not hamper the memory cell's second (two-phase) R communication (see #`latch(L)`# in Section IV-C), as [~`ri`], which corresponds to `so`− (see #`yctl(T)`# in Section IV-D), becomes true at the beginning of the burst. Thus, the memory-cell can complete its second G communication with the arbiter-interface right away.

The read sequence above yielded the following PRS:

```
di&~t->d+    ~di->d-
d&t->do+    ~d|~t->do-.
```

Weakening `do`−'s guard is safe because `t` becomes false after `d` does. The circuit is shown in Fig. 10(a).

And we compiled the final reshuffling of the write sequence (see #`yctl(T)`# in Section IV-D) into the following PRS:

```
       ~ti&d&si->t+,tro+,so-
ti&~d&~si&~tco->t-,tro-,so+.
```

To prevent `tro`− from firing before the last column-address transmission is completed (see below), we strengthened its guard with ~`tco`. This precaution is necessary because when the column arbiter interface is acknowledged (`tco`+) it clears its acknowledge to the memory cell, at which point `si` becomes false (see Fig. 6). Then, `tro`− could fire while we are waiting for `ti` to become false in response `tco`+ (see below). The circuit is shown in Fig. 11(a).

For MUX(b, x)'s column communications, $T!(C?)$, making port C passive and port T active yielded this HSE sequence

```
*[[ci];co+;[˜ci];co-;[ti];tco+;[˜ti];tco-].
```

Relocating the T communication's two halves a quarter and three-quarters of the way through the C communication yielded the reshuffling presented in Section IV-D (#xctl(T)#). Thus, reception is acknowledged (co+) at the same time transmission starts (tco+), which requires us to make the encoder's outputs state-holding (see Fig. 6). Therefore, we added staticizers to all its outputs, including the extra always-a-one line that serves as a request, and we tied a pFET to the request line—as in Fig. 8—to clear it.

We compiled the final reshuffling into the following PRS:

```
ti&ci&tro->co+,tco+    ˜ti&˜ci->co-,tco-.
```

We have strengthened the guard of tco+ with tro to block a column address from a newly loaded row from being transmitted while we are waiting for ti to clear, after tro goes low. The circuit is shown in Fig. 11(b).

ACKNOWLEDGMENT

The author would like to thank C. Higgins, T. Horiuchi, B. Linares-Barranco, and T. Serrano-Gotarredona for their invaluable help in beta-testing this interface, and fishing out and documenting bugs. He would also like to thank P. Merolla for helping with adding serial-address transmission to the design.

REFERENCES

[1] C. A. Mead and T. Delbruck, "Scanners for visualizing analog vlsi circuitry," *Analog Integ. Circuits Signal Process.*, vol. 1, pp. 93–106, 1991.

[2] W. Yang, "A wide-dynamic range low-power photosensor array," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC'94)*, vol. 37, San Francisco, CA, 1994, p. 230.

[3] B. Fowler, A. E. Gamal, and D. Yang, "A CMOS area image sensor with pixel-level A/D conversion," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC'94)*, vol. 37, San Francisco, CA, 1994, pp. 226–227.

[4] L. G. McIlrath, "A low-power low-noise ultrawide-dynamic-range cmos imager with pixel-parallel A/D conversion," *IEEE Trans. Solid-State Circuits*, vol. 36, pp. 846–853, May 2001.

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

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

[7] K. A. Boahen, "The retinomorphic approach: pixel-parallel adaptive amplification, filtering, and quantization," *Analog Integr. Circuits Signal Process.*, vol. 13, pp. 53–68, 1997.

[8] E. Culurciello, R. Etienne-Cummings, and K. Boahen, "Arbitrated address event representation digital image sensor," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC'01)*, Feb. 2001, pp. 92–93.

[9] J. Kramer, "An on/off transient imager with event-driven asynchronous readout," in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. 2, 2002, pp. II-165–II-168.

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

[11] M. Sivilotti, "Wiring considerations in analog VLSI systems, with application to field-programmable networks," Ph.D. dissertation, Dept. Comp. Sci., California Institute of Technology, Pasadena, CA, 1991.

[12] A. Mortara, E. Vittoz, and P. Venier, "A communication scheme for analog VLSI perceptive systems," *IEEE J. Solid-State Circuits*, vol. 30, pp. 660–669, June 1995.

[13] A. Abusland, T. S. Lande, and M. Hovin, "A VLSI communication architecture for stochastically pulse-encoded analog signals," in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. 3, May 1996, pp. 401–404.

[14] K.A. Boahen, "Communicating neuronal ensembles between neuromorphic chips," in *Neuromorphic Systems Engineering: Neural networks in Silicon*, T. S. Lande, Ed. Boston, MA: Kluwer Academic, 1998, ch. 11, pp. 229–262.

[15] ——, "Point-to-point connectivity between neuromorphic chips using address-events," *IEEE Trans. Circuits Syst. II*, vol. 47, pp. 416–434, May 2000.

[16] C. A. Mead, *Introduction to VLSI Systems*. Reading, MA: Addison Wesley, 1980.

[17] J. G. Elias, "Artificial dendritic trees," *Neural Computation*, vol. 5, pp. 648–663, 1993.

[18] S. R. Deiss, R. J. Douglas, and A. M. Whatley, "A pulse-coded communications infrastructure for neuromorphic systems," in *Pulsed Neural Networks*, W. Maass and W. B. C. M, Eds. Boston, MA: MIT Press, 1999, ch. 6, pp. 157–178.

[19] C. M. Higgins and C. Koch, "Multi-chip motion processing," in *Proceedings of Conference on Advanced Research in VLSI*. Los Alamitos, CA: IEEE Comp. Soc. Press, 1999, vol. 20, pp. 309–322.

[20] S. P. DeWeerth, G. N. Patel, M. F. Simoni, D. E. Schimmel, and R. L. Calabrese, "A VLSI architecture for modeling intersegmental coordination," in *Proc. 17th Conf. Advanced Research in VLSI*, 1997, pp. 182–200.

[21] J. P. Lazzaro and J. Wawrzynek, "A multi-sender asynchronous extension to the address-event protocol," in *Proc. 16th Conf. Advanced Research in VLSI*, 1995, pp. 158–169.

[22] K. A. Boahen, "A throughput-on-demand address-event transmitter for neuromorphic chips," in *Proc. 20th Anniversary Conf. Advanced Research in VLSI*, 1999, pp. 72–86.

[23] ——, "A burst-mode word-serial address-event link II: Receiver design," *IEEE Trans. Circuits Syst. I*, vol. 51, pp. 1281–1291, July 2004.

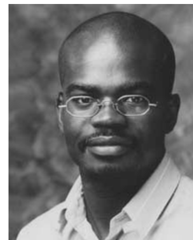
[24] ——, "A burst-mode word-serial address-event link—III: Analysis and test results," *IEEE Trans. Circuits Syst. I*, vol. 51, pp. 1292–1300, July 2004.

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

[26] M. Schwartz, *Telecommunication Networks: Protocols, Modeling, and Analysis*. Reading, MA: Addison-Wesley, 1987.

[27] K. A. Boahen, "Retinomorphic vision systems II: communication channel design," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 1996, pp. 14–17.

[28] A. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Proceedings of UT Year of Progamming Institute on Concurrent Programming*. Reading, MA: Addison-Wesley, 1990, pp. 1–64.



Kwabena A. Boahen received the B.S. and M.S.E. degrees in electrical and computer engineering from The Johns Hopkins University, Baltimore, MD, in the concurrent masters-bachelors program, both in 1989, and the Ph.D. degree in computation and neural systems from the California Institute of Technology, Pasadena, in 1997.

He is an Associate Professor in the Bioengineering Department at the University of Pennsylvania, Philadelphia, where he holds a secondary appointment in electrical engineering. His current research interests include mixed-mode multichip VLSI models of biological sensory and perceptual systems, and their epigenetic development, and asynchronous digital interfaces for interchip connectivity.

Dr. Boahen was awarded a Packard Fellowship in 1999, a National Science Foundation CAREER Grant in 2001, and an Office of Naval Research YIP Grant in 2002. He is a member of Tau Beta Kappa and has held a Sloan Fellowship for Theoretical Neurobiology at the California Institute of Technology.