July 2004

A burst-mode word-serial address-event link--II: receiver design

Kwabena A. Boahen
University of Pennsylvania, boahen@seas.upenn.edu

A burst-mode word-serial address-event link--II: receiver design

Abstract
We present a receiver for a scalable multiple-access inter-chip link that communicates binary activity between two-dimensional arrays fabricated in deep submicron CMOS. Recipients are identified by row and column addresses but these addresses are not communicated simultaneously. The row address is followed sequentially by a column address for each active cell in that row; this cuts pad count in half without sacrificing communication capacity. Column addresses are decoded as they are received but cells are not written individually. An entire burst is written to a row in parallel; this increases communication capacity with integration density. Rows are written one by one but bursts are not processed one at a time. The next burst is decoded while the last one is being written; this increases capacity further. 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 submicron CMOS technology.

Keywords
asynchronous logic synthesis, event-driven communication, neuromorphic systems, pipelining, pixel-level quantization, serial-to-parallel conversion

A Burst-Mode Word-Serial Address-Event Link—II: Receiver Design

Kwabena A. Boahen

***Abstract*—We present a receiver for a scalable multiple-access inter-chip link that communicates binary activity between two-dimensional arrays fabricated in deep submicron CMOS. Recipients are identified by row and column addresses but these addresses are not communicated simultaneously. The row address is followed sequentially by a column address for each active cell in that row; this cuts pad count in half without sacrificing communication capacity. Column addresses are decoded as they are received but cells are not written individually. An entire burst is written to a row in parallel; this increases communication capacity with integration density. Rows are written one by one but bursts are not processed one at a time. The next burst is decoded while the last one is being written; this increases capacity further. 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 submicron CMOS technology.**

***Index Terms*—Asynchronous logic synthesis, event-driven communication, neuromorphic systems, pipelining, pixel-level quantization, serial-to-parallel conversion.**

I. Scaling Two-Dimensional Arrays

EVENT-DRIVEN demultiplexers are used to deliver binary signals to arrays of parallel-processing cells. Traditionally, clock-driven demultiplexers were used for this purpose. However, updating each and every cell regularly is wasteful if activity is sparse, either in time or in space. In that case, it is more efficient to update a cell only when its input changes, which may be accomplished simply by delivering the cell's address to the array. A decoder then selects the cell and either toggles its input (level coded) or drives the input high briefly (pulse coded). This **address-event** representation has been used to communicate pulse-coded outputs of silicon retinae [1]–[4] and cochleas [5], to drive arrays of silicon neurons [1], [2], [6], [7], and to interface these neuromorphic chips with computers [5]. Interest in event-driven multiplexer-demultiplexer links is increasing, driven by the trend toward quantizing signals inside the array (e.g., active pixel sensors [8]–[10] and pulse-coded neural networks [11], [12]).

We recently developed an event-driven multiplexer that boosts capacity by reading an entire row of cells in parallel [13]. As feature sizes shrink, it takes longer to cycle the row and column lines because faster logic (minimum-sized inverter chain) is neutralized by larger load (cells per row or column). This bottleneck limits prior multiplexer designs, where a single active cell is read at a time [11], [14]–[16]. We broke the bottleneck by exploiting parallelism—reading an entire row simultaneously. Communication capacity is not compromised when we serially encode the addresses of active cells in that row because we use devices much larger than those in the array. As communication capacity is boosted without sizing up devices inside the array, our multiplexer design can exploit the high integration densities deep submicron processes offer.

In this paper, we describe a complementary event-driven demultiplexer that writes an entire row of cells in parallel. Prior demultiplexer designs write a single cell at a time [1], [2], similar to prior multiplexer designs. Hence, they also face a bottleneck. Our new demultiplexer design provides a scalable solution when paired with our recently developed multiplexer design to form a parallel read–write link. The increase in parallelism as the array gets denser enables the communication capacity to keep increasing, despite the fact that faster logic is neutralized by larger load. Our demultiplexer requires large devices only in the periphery, where serial-to-parallel conversion occurs. The entire parallel read–write link is implemented asynchronously—the same approach adopted by prior link designs—to facilitate its use in large heterogeneous multichip systems.

Similar to prior designs, our event-driven multiplexer-demultiplexer link provides *virtual* connectivity between cells in the same array or in different arrays, which need not be on the same chip. That is, the multiplexer, or transmitter, uses an encoder to generate an address that uniquely identifies an event's place of origin. Conversely, the demultiplexer, or receiver, uses a decoder to recreate the event at the destination [1], [5], [14]. These virtual wires can be rerouted by using a look-up table to translate an incoming address into one or more outgoing addresses [7], [17], [18]. Events may be fanned out to multiple receivers by using splits and merges [6], or with a shared bus [18], [19]. Thus, in addition to providing point-to-point communication for parallel distributed processing in multi-chip systems, the single-transmitter-single-receiver address-event link described here can serve a wide variety of purposes when augmented appropriately.

However, unlike prior designs, our event-driven multiplexer–demultiplexer link communicates row and column addresses serially, rather than in parallel. By going serial, we cut the pad count in half—without sacrificing communication capacity. There is no loss in communication capacity because the multiplexer does not retransmit the row address if the next event is from the same row. These events are communicated in a burst:

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.2004.830702

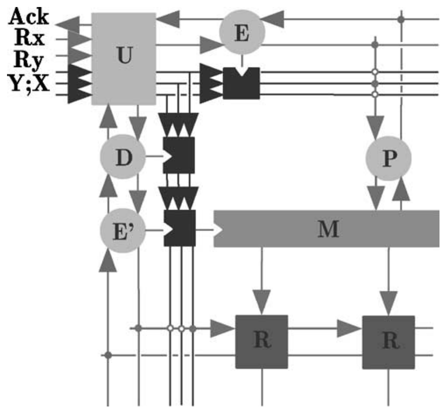


Fig. 1. Receiver architecture: a two-way demultiplexer (U) directs row addresses to one latch (D) and column addresses to another (E). When the burst ends, the row's address and its decoded column addresses (P) are written to a second set of latches (E′ and M). As the row address is decoded and the column data is written to that row (R), the next burst is received and its column addresses are decoded.

the row address, a column address for each active cell, and a termination signal. It is obvious which events are from the same row since an entire row of cells is read out in parallel. Thus, parallel readout makes it possible to eliminate redundant row addresses and thereby communicate addresses serially without sacrificing capacity.

The paper is divided into four sections. In Section II, we present a high-level specification for the receiver, and decompose it into a hierarchy of concurrent processes. In Section III, we present the final handshaking sequences and the resulting asynchronous logic circuits; intermediate synthesis steps can be found in the Appendix. Section IV concludes the paper. A parallel-read burst-mode transmitter and analysis and test results are presented in companion papers [13], [20].

II. Receiver Design

Our goal is to optimize the address-event receiver's row–column architecture in three ways. For an n-cell array, the following hold.

1) Multiplexing row and column addresses cuts pad count in half.
2) Pipelining serial-to-parallel conversion increases communication capacity.
3) Writing a row's events in parallel boosts communication capacity by up to $\sqrt{\mathsf{n}}$.

As alluded to in Section I, we realize Optimization 1 by eliminating redundant row addresses, thus the drop in (peak) capacity is only 1 in $\sqrt{\mathsf{n}}+1$.We realize Optimization 2 by decoding the new row's column addresses even while we are writing the previous row's events into the array. Finally, we realize Optimization 3 by writing up to $\sqrt{\mathsf{n}}$ events in parallel assuming a square array.

A preview of the receiver 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 [21] (tutorial examples are provided in [22]). His methodology involves applying a series of program

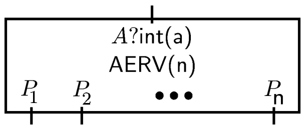


Fig. 2. Receiver specification: the receiver inputs an a-bit address on port A and communicates on the corresponding one of its n dataless ports P_j.

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 processes behave when executed in parallel, which is extremely difficult. After decomposing the receiver specification into a set of concurrent programs, we compile these one-line programs into hardware in Section III.

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 [21]. In CHP, logic circuits "execute" concurrent programs, for example

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

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 $\mathsf{RLY(n)}$ 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 receiver is shown in Fig. 2. We use **selection**, $[G_1 \rightarrow S_1 [\!] G_2 \rightarrow S_2 [\!] \cdots [\!] G_\mathsf{n} \rightarrow S_\mathsf{n}]$, to choose the recipient. This program construct picks a guard G_j that is true and executes the corresponding program segment S_j.[1] In our case, the guard is a single bit of d, an n-bit word obtained by decoding an a-bit address received on port A, the receiver's input, where $\mathsf{a} = \log_2(\mathsf{n})$. And the program segment communicates on one of the receiver's n dataless ports, P_j, to signal the occurrence of an event. Thus, we have

$$\mathsf{AERV(n)} \equiv *[d := \mathsf{dec}(A?); [d.1 \rightarrow P_1 [\!] d.2 \rightarrow P_2 [\!] \cdots [\!] d.\mathsf{n} \rightarrow P_\mathsf{n}]]$$

where $d.j$ is the jth bit of d, which is assigned (:=) the value returned by calling $\mathsf{dec}(\cdot)$ with the address read from port A.

[1]If all the guards are false, it waits for one to become true; they must be mutually exclusive.

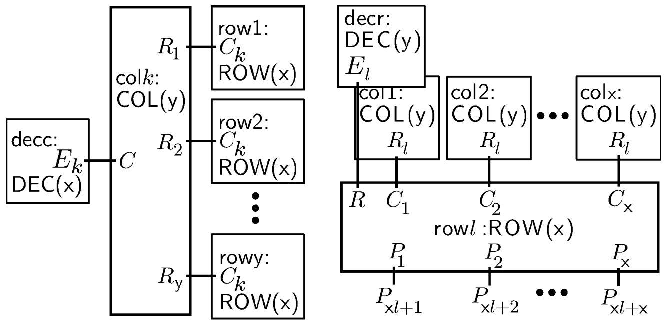


Fig. 3. Row–column organization. (a) The kth column communicates with all the rows (row$l$) as well as the column decoder (decc). (b) The lth row communicates with the row decoder (decr) and all the columns (col$k$). It services event recipients x$l+1$ through x$(l+1)$ with its P_k ports.

This function converts a binary code (a-bit) into a one-hot one (n-bit).

Alternatively, the receiver process may be described succinctly using CHP's **replication** construct: $\langle \diamond j : 1 \, . \, . \, \mathsf{n} : S_j \rangle \equiv S_1 \diamond S_2 \diamond \cdots \diamond S_{\mathsf{n}}$, where each S_j is a program-segment and $\diamond$ is any operator that can be concatenated. As the selection operator ($[\!]$) can be concatenated, we have

$$\mathsf{AERV(n)} \equiv *[d := \mathsf{dec}(A?); [\langle [\!] j : 1 \, . \, . \, \mathsf{n} : d.j \rightarrow P_j \rangle]] \, .$$

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 : AERV(n) specifies an instance of AERV(n) named aer) and their ports' **data types** (e.g., A!int(8) specifies that port A outputs bytes). Ports that are defined neither as input nor 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 AERV(n) into separate row, column, and decoder processes, named ROW(x), COL(y), and DEC(m), respectively. These processes are connected as shown in Fig. 3. This decomposition is accomplished through three program transformations. For the first transformation, we reorganize AERV's n dataless ports into y rows and x columns and replace its input, port A, with two inputs, ports A_1 and A_2, that accept row and column addresses, respectively. We use a 1-in-y decoder to select a row and a 1-in-x decoder to select a cell in that row. Thus, we have

$$\begin{aligned}\mathsf{AERV(y,x)} \equiv *[&r := \mathsf{dec}(A_1?) \| c := \mathsf{dec}(A_2?); \\ &[\langle [\!] l : 1 \, . \, . \, \mathsf{y} : r.l \rightarrow [\langle [\!] k : 1 \, . \, . \, \mathsf{x} : c.k \rightarrow P_{l \cdot k} \rangle] \rangle]]\end{aligned}$$

where $\mathsf{y} \times \mathsf{x} = \mathsf{n}$ and $l \cdot k = \mathsf{x}l + k$. Parallel lines ($\|$) denote **parallel** execution. The decoded addresses are stored in local y-bit and x-bit words named r and c, respectively.

For the second transformation, we implement address decoding in a separate process, DEC(m). This 1-in-m decoder uses a $\log_2(\mathsf{m})$-bit input, port L, for the address, a local m-bit word w for the decoded address, and m dataless ports (E_j), for selection. That is

$$\mathsf{DEC(m)} \equiv *[w := \mathsf{dec}(L?); [\langle [\!] j : 1 \, . \, . \, \mathsf{m} : w.j \rightarrow E_j \rangle]] \, .$$

Two instances of DEC(m), with $\mathsf{m} = \mathsf{y}$ or x, are used for row and column decoding, respectively.

Having removed the decoders, we are left with a process containing just the array of dataless ports, which we call ARY(y, x). This process uses y dataless ports (R_l) and x dataless ports (C_k) to communicate with the row and column decoders, respectively. It probes these ports to find out which of its rows or columns has been selected. The **probe**, $\overline{P_j}$, evaluates to true when there is a communication pending on port P_j (i.e., the other process is waiting). Having found the selected row and column, ARY(y, x) communicates on the corresponding dataless port and communicates with the column and row decoders to acknowledge its selection. Thus, we have

$$\begin{aligned}\mathsf{ARY(y,x)} \equiv \; *[[\langle [\!] l &: 1 \, . \, . \, \mathsf{y} : \\ &\overline{R_l} \rightarrow \; [\langle [\!] k : 1 \, . \, . \, \mathsf{x} : \overline{C_k} \rightarrow P_{l \cdot k} \| C_k \rangle] \, \| \, R_l \rangle]].\end{aligned}$$

For our third and final transformation, we break up ARY(y, x) into x column processes and y row processes. As shown in Fig. 3, COL(y) communicates with the column decoder using its C port while ROW(x) communicates with the row decoder using its R port; these ports are dataless. COL(y) also performs a column-wide broadcast on its R_l ports, which are connected to the rows' C_k ports. A **broadcast**, denoted by the circle ($\circ$), waits for at least one recipient to respond.[2] Thus, programs for the column and row processes read

$$\begin{aligned}\mathsf{COL(y)} &\equiv *[C; \langle \circ l : 1 \, . \, . \, \mathsf{y} : R_l \rangle] \\ \mathsf{ROW(x)} &\equiv *\left[R; \left[\langle [\!] k : 1 \, . \, . \, \mathsf{x} : \overline{C_k} \rightarrow P_k \| C_k \rangle\right]; R\right] .\end{aligned}$$

The second R communication is included to prevent the row decoder from selecting another row until the column communication is finished. A second communication must be added to DEC(m) as well to reflect this. With mutual exclusion guaranteed, it is no longer necessary to synchronize the row and column decoders; they can read and decode addresses independently.

This serial-write architecture, where cells are written individually, was first implemented in [1]; it was pipelined by adding latches to the decoder inputs in [23] and [24]. We design a parallel-write version in the next subsection.

C. Writing the Array in Parallel

Here, we modify COL(y) and ROW(x) to write all events destined for the same row in parallel, an innnovation introduced in this work. These events are received in a burst, the row address followed by a column address for each active cell [13]. Hence, we must demultiplex the row and column addresses and detect when the burst ends, signaled by a reserved address named ϕ, only then can the parallel write begin. We introduce a process named DMX(b, x) to perform these tasks, where $\mathsf{b} = \max(\log_2(\mathsf{x}), \log_2(\mathsf{y}))$. Predecoded column addresses are

[2]This construct is not supported by CHP; its use is discouraged as there is no delay-insensitive implementation. The reason is that communication signals sent to inactive recipients are not acknowledged, and therefore, we cannot tell if they have been cleared.

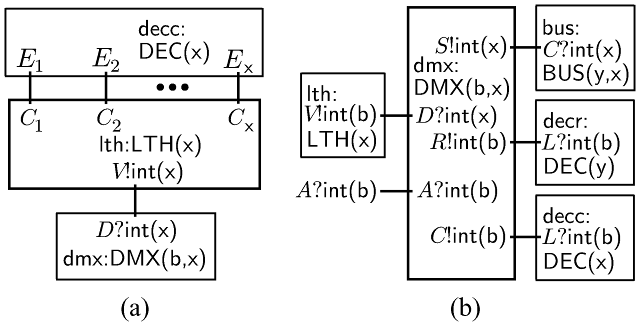


Fig. 4. Burst reception. (a) Latch (lth) is written to by the column decoder (decc) and read by the demultiplexer (dmx). (b) Demultiplexer (dmx) relays words from the latch (lth) to the bus (bus). It also receives row and column addresses from the receiver's global A port and sends them to the appropriate decoder (decr and decc).

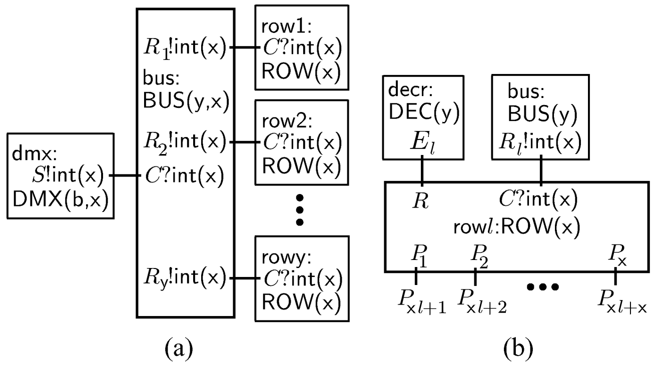


Fig. 5. Parallel write-in. (a) Bus (bus) transfers x-bit words handed off by the demultiplexer (dmx) to the selected row (row$l$). (b) Row's C_k ports have been combined into a single port C that inputs x-bit words.

stored in a x-bit latch, named LTH(x). These processes are connected as shown in Fig. 4.

For LTH(x) [see Fig. 4(a)], we simply **set** the kth bit ($w.k\uparrow$ or $w := w + 2^{k-1}$) of its x-bit word (w) when the kth column is selected (C_k) and **clear** every bit ($w.k\downarrow$ or $w := 0$) after the latch is read ($V!w$). Thus, its program reads

$$\mathsf{LTH(x)} \equiv \ast\big[\big[\langle [\,]k : 1\,..\,\mathsf{x} : \overline{C_k} \rightarrow w.k\uparrow \| C_k\rangle \\ [\,]\overline{V} \rightarrow V!w; w := 0\big]\big].$$

LTH(x) is read ($V!w$) by DMX(b, x) at the end of the burst, after all the column addresses have been decoded [see Fig. 4(a)]. Hence, we assumed that the V communication does not overlap with any of the C_k communications, allowing us to include it in the same selection statement. This mutual exclusion can be guaranteed by DMX(b, x), as we shall show next.

For DMX(b, x) [see Fig. 4(b)], we read bursts from the receiver's word-serial input (port A), direct the first address to the row decoder (port R) and direct subsequent addresses to the column decoder (port C), until we receive ϕ. When that happens, we transfer LTH(x)'s data to the array ($S!(D?)$) and repeat the procedure. Thus, we have

$$\mathsf{DMX(b,x)} \equiv \ast[A?a; [a \neq \phi \rightarrow C!a [\,] a = \phi \rightarrow R!d \| S!(D?); A?d]]$$

where a and d are local b-bit words. We chose to delay passing on the row address ($R!d$) until the burst ends ($a = \phi$). This rearrangement allows us to use a single control signal to initiate row selection as well as column data transfer ($S!(D?)$), since, at this point, all the column addresses have been decoded. Note that the row address is the one that follows ϕ; it is stored in d. Upon initialization, execution must begin at this $A?d$ communication.

To write LTH's data into the selected row, we combine our x COL(y) processes into a single x-bit-wide bus, named BUS(y, x), as shown in Fig. 5(a). And we make ROW(x) compatible by combining its x dataless C_k ports into a single x-bit input, port C, as shown in Fig. 5(b). We use **concurrency**, $[G_1 \rightarrow S_1, G_2 \rightarrow S_2, \cdots, G_{\mathsf{n}} \rightarrow S_{\mathsf{n}}]$, to update all the cells selected in that row. This construct executes (concurrently) all program segments, S_j, whose guards, G_j, are true.[3] Thus, we have

$$\mathsf{BUS(y,x)} \equiv \ast[C?w; \langle \circ l : 1\,..\,\mathsf{y} : R_l!w\rangle]$$
$$\mathsf{ROW(x)} \equiv \ast[R; C?w; [\langle , k : 1\,..\,\mathsf{x} : w.k \rightarrow P_k; w.k\downarrow\rangle]; R]$$

where w is a x-bit word that is local to each process.

In summary, we have decomposed AERV(n) into five concurrent processes: One instance of DMX(b, x), two of DEC(n), one of LTH(x), one of BUS(y, x), and y of ROW(x). These process' ports are connected together as shown in Figs. 4 and 5. The next step in the synthesis procedure is to compile these CHP programs into hardware.

III. Receiver 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]This construct is not supported by CHP. Its use is discouraged because termination cannot always be guaranteed, but that is not the case here. Concurrency waits for at least one guard to become true if necessary, just like selection does.

3) **Dual-rail** acheives 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 is 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 [21], we make two choices when we perform HSE. First, we make output ports active and input ports passive.[4] 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 $\rightarrow$ b− clears a bit b when a boolean expression e becomes true. We write ~e $\rightarrow$ b+ to set the bit when the expression is false. An n-type field-effect transistor (nFET) implements the former rule while a p-type field-effect transistor (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 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. Demultiplexing

The CHP program for DMX(b, x) [see Section II-C and Fig. 4(b)] requires us to read addresses from the receiver's input (port A), store the row address, pass column addresses on to the column decoder (port C), and then pass on the row address (port R) and transfer the row word (port D to port S) when the burst ends. These operations are implemented in this section; we implement BUS(y, x) here as well.

We use the passive counterpart of the transmitter's three-wire protocol [13] for port A. That is, we use bundled-data but we

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

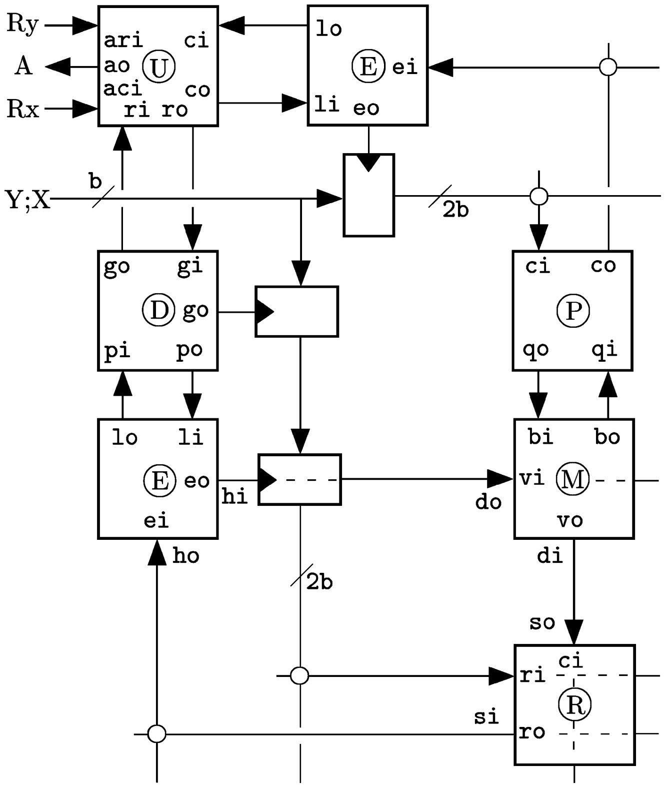


Fig. 6. Receiver schematic. DMX consists of a two-way demultiplexer (U) that parses b-bit row and column addresses (Y;X), and a latch (controlled by block D) that holds the row address till the burst ends. DEC (two instances) includes a latch (controlled by block E) that drives its decoding logic (represented by discs) with dual-rail encoded data (2b lines). LTH's bit cells consist of a buffer (P) and a memory (M). DMX includes a set of wires that connect LTH's cells (do, di), to the array (so, si), and to a controller (ho, hi). These wires broadcast column data to ROW's cells (R), thereby implementing BUS as well.

distinguish row and column addresses using separate request lines, ari and aci, respectively; they share the same acknowledge, ao. Fig. 6 shows the correspondence between these signals and the receiver's Ry, Rx, and Ack signals mentioned earlier. The sequence of waits and actions on these three lines is

```
*[[ari];ao+;[~aci];ao-]
*[[aci];ao-;[~aci];ao+]
```

where || denotes parallel execution, just like in CHP. The first sequence's first half receives the row address, while the second half receives the burst-termination signal ϕ. Multiple column addresses are received by executing the second sequence as many times as desired, halfway through the first sequence. This three-wire handshake is illustrated in Fig. 7. The address switches back to the row address because the transmitter's address mux is controlled by the column request (i.e., aci). Note that ari remains high throughout.

To demultiplex addresses received on DMX's port A (passive), we augment the three-wire handshaking sequence above with communications on ports R and C (both active), which are connected to the row and column decoder, respectively [see

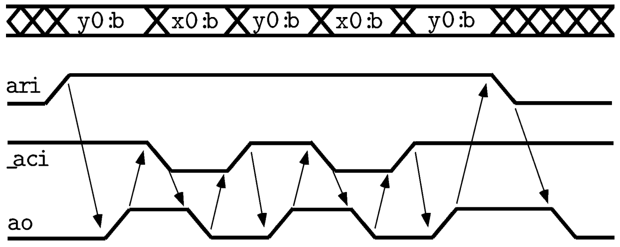


Fig. 7. Three-wire handshake: when ari becomes high, receiver reads the row address y0 : b and raises ao. When _aci becomes low, the receiver reads the column address x0 : b and lowers ao. This column communication is completed by taking _aci high—the address reverts back to y0 : b—followed by ao. After a second column address is received, the burst is terminated by taking ari low, followed again by ao.

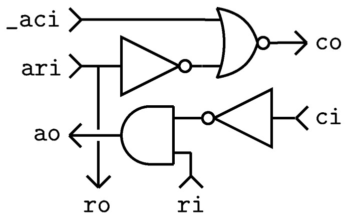


Fig. 8. Three-to-four-wire converter [U]. Requests ari and _aci are relayed to the decoders on ro and co, while acknowledges ri and ci are merged onto ao.

Fig. 4(b)]. Thus, we obtained the following reshuffled HSE sequence:

```
# demux (U) #
*[[ari];ro+;[ri];ao+;[~ari];ro-;[~ri];ao-]
  ||*[[aci];co+;[ci];ao-;[~aci];co-;[~ci];ao+].
```

These sequences convert the three-wire handshake into a four-wire one, with separate acknowledges for row and column addresses. The second two-phase communication on port R now signals the end of a burst.

Compiling this HSE into PRS (see Appendix part A) yielded the circuit shown in Fig. 8. When ari becomes high, ro goes high, which prompts ri to become high. Both of the AND gates' inputs are now high, so it drives ao high. Row-address reception is now complete. Column-address reception starts with _aci becoming low. Both of the NOR gates' inputs are now low, so it drives co high, which prompts ci to become high. Hence, the inverter's output goes low, which forces the AND gate's output ao low. Column-address reception is now complete. Now, initial states must be restored. co is cleared once _aci goes back high and ao returns to the high state once ci becomes low. ro is cleared next, once ari swings low, and ao is cleared once ri becomes low.

We do not decode the row address, which is read on the first (two-phase) R communication, until the second R communication occurs (see #demux(U)# above). This delay is realized by the following reshuffled HSE sequence (see Appendix part B), which communicates with the three-to-four-wire converter on its G port (passive) and communicates with the row decoder on its P port (active), as shown in Fig. 6

```
# delay (D) #
*[[gi];go+;[~gi&~pi];po+;[pi];go-;po-].
```

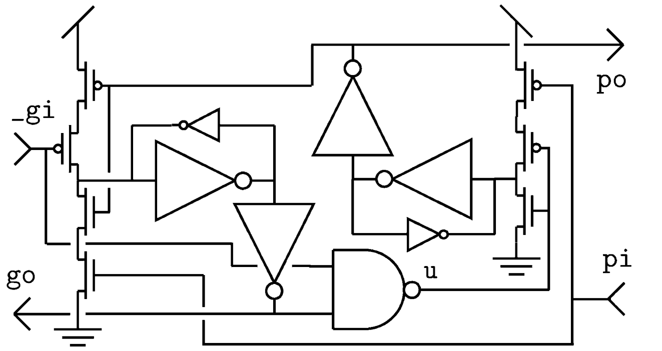


Fig. 9. Row-address delay [D]. Transfers row address from three-to-four-wire converter (_gi, go) to row decoder (po, pi), after holding it in a latch (go) till the burst ends.

As required, P does not begin until the second half of G starts, which signals the end of a burst. At this point, all the column addresses have been decoded, and a parallel write may begin as soon as the row address is decoded.

Compiling this HSE into PRS (see Appendix part B) yielded the circuit shown in Fig. 9. Initially, po is low, so go goes high as soon as _gi becomes low. After _gi goes back high, u swings low, since both inputs to the NAND gate are now high. u becoming low will make po go high, provided pi is low. go also strobes the latch that holds the row address (described in Section III-B). The latch becomes opaque when go is high. It does not become transparent again until go goes low, after pi becomes high, which signals that the address has been read. go becoming low also clears po by forcing u high.

DMX's parallel read–write operation ($S!(D?)$) is realized by the following reshuffled HSE sequence, which uses a straight-data representation. Ports S and D are both active; a third port, H, which is passive and dataless, gives the go signal.

```
# transfer #
*[[hi];do+;[di];so+;[si];ho+;
[~hi];do-;[~di];so-;[~si];ho-].
```

This sequence is implemented by three wires: tie hi to do, di to so, and si to ho. BUS may also be implemented with wires [see Fig. 5(a)]: these connect its passive input (port C) to its active outputs (ports R_l). Thus, the parallel write is realized simply by extending the so lines (one per bit) into the array (x wires in all), connecting them to every row.

The triangular communication that implements the parallel read–write operation is shown in Fig. 6. It is controlled by the delay circuit (D), whose P port drives port H. Thus, column data transfer and row-address decoding are initiated simultaneously. The E block in this figure was inserted to pipeline the decoder; this is described in the next subsection (Section III-B). Ignoring E for the moment, and imagining that li is tied directly to eo and ei to lo, we observe a triangular path connecting the delay circuit (D), the column data latch (M), and the receiving row (R). The receiving row's acknowledge (si), which signals column parallel write as well as row selection (see Section III-C), is fed to a y-input OR gate (part of the decoder) that combines row acknowledges into a single array-level acknowledge (ho).

This completes our implementation of DMX(b, x) as well as BUS(y, x), which we have essentially folded into DMX(b, x).

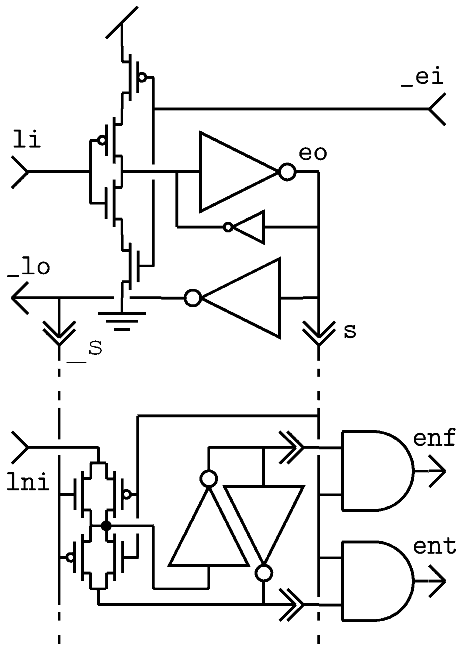


Fig. 10. Address latch [E]. Interfaces three-to-four-wire converter or row-address delay (`li`, `lni`, `_lo`) with decoder-logic (`ent`, `enf`, `_ei`). The latch is opaque when its control signal (`s`) is high. Dual-rail encoding is used for data output.

B. Decoding

We now turn our attention to the CHP program for DEC(m) (see Section II-B). In addition to the combinational logic required to implement the binary to one-hot function, dec(·), our decoder design includes a latch, to support pipelined operation, and a bundled-data-to-dual-rail converter, to guarantee delay-insensitive operation. Here we describe the latch and the converter; the combinational logic was described in [22]. There is also an m-input OR gate (mentioned above), built from a tree of two-input ORs, which combines the m acknowledges.

We made the latch's input (port L) passive and its output (port E) active and used the following reshuffled HSE sequence:

```
# pipeline (E) #
*[[~ei&li];eo+,lo+;[ei&~li];eo-,lo-].
```

This reshuffling allows the decoder to acknowledge receiving the address before it has finished decoding it, thereby increasing throughput [23], [24]. The compiled circuit, called a **C-element** [25], is shown in Fig. 10, which also includes the memory cell. The C-element clears `_lo` and sets `eo` when `li` and `_ei` are both high. Thus, the write is acknowledged and a read is initiated at the same time. It sets `_lo` and clears `eo` when `li` and `_ei` are both low, thus enabling a new write and terminating the old read at the same time.

The latch's memory cell consist of two inverters and a mux, whose select signal `s` is driven by the C-element. When `eo` goes high, `s` is driven high and `_s` is driven low. Thus, the latch becomes opaque. It does not become transparent again until `eo` goes low, which happens after `_ei` is lowered, indicating that the address has been decoded.

The data converter consist of two AND gates that combine the control signal (i.e., `eo` or `s`) with the stored bit and its complement. Thus, either `ent` or `enf` is forced high depending on the nth bit's value when `eo` goes high. One or the other of these dual rails is connected to the $\log_2(m)$-input AND gates that generate the decoder's m (one-hot) outputs, depending on that particular address [22]. Thus, when all $\log_2(m)$ bits are valid, an output will become active. If even just one bit is invalid (i.e., both rails are low), all the outputs will remain inactive.

Previously, we drove the decoding logic with bundled-data instead of dual-rail. That is, we connected one or the other of the memory cell's complementary outputs directly to each $\log_2(m)$-input-AND gate, which had an extra input driven by the request (i.e., `eo`) that served as an enable [22]. We abandoned this scheme as we found that delays on this enable line produced glitches, because the $\log_2(m)$-input AND gates would remain enabled for some time after `eo` was driven low. Since the latch becomes transparent at this point, new data would propagate straight through and drive the AND gates until the delayed enable was cleared, hence the glitch. Glitching is prevented by using dual rail, which eliminates the extra line and logic level as well.

We use the same pipelined decoder design for both the row and column addresses. For the row decoder, this means that the parallel write is also pipelined, as the C-element's `eo` signal initiates column data transfer as well as row selection (see Fig. 6). Hence, when the delay circuit passes on the row address at the end of the burst, it will be acknowledged at the same time we begin transferring the data into the array. This simultaneity allows the delay circuit to complete its G communication with the three-to-four-wire converter early (see Section III-A). Hence, the next burst can be received, and its column addresses decoded, even while we are performing the parallel write. The column data latch, which is presented in the next subsection (Section III-C), must be designed to ensure that the old data is not overwritten.

C. Writing

We implement LTH(x) and ROW(x) in this section. LTH(x) must complete each and every C_k communication in its entirety before it has even begun one V communication [see Fig. 4(a)]. This ability for communications on one port to get out of step with communications on another is referred to as **slack**. For instance, in a regular left–right buffer, like that used to pipeline the decoders (see Section III-B), the second phase of the left port's communication starts at the same time as the first phase of the right port's. Thus, a little more than a quarter cycle of slack is provided. For LTH's cells, a full cycle of slack is required to allow another burst to be read while we are writing the preceding one into the array.

To obtain a full cycle of slack, we implement the LTH cell in two stages called column buffer and data latch, each of which contributes at least half a cycle of slack. We made the port C that column buffer uses to communicate with the decoder passive (see Section II-C). We made the port Q it uses to communicate with data latch active (see Fig. 6). Buffer's reshuffled bit-level HSE sequence reads (see Appendix part C)

```
# buffer (P) #
*[[ci];co+;[~qi];qo+;[~ci];co-;[qi];qo-].
```

C finishes before the request to the second stage has even been acknowledged (i.e., [qi]). In fact, as [˜qi] is from the end of the previous cycle, only the first of Q's four phases overlaps with C. Hence, buffer provides a slack of three-quarters.

Compiling buffer's HSE into PRS (see Appendix part C) yielded the circuit shown in Fig. 11(a). co goes high as soon as ci becomes high, provided qo is low. co becoming high will make qo go high, provided qi is low. And a high qo will clear co after ci goes back down. With co low, qi becoming high is all that is required to clear qo. Essentially, this circuit is just two serially connected C-elements, like those used to control the address-latch (see Fig. 10).

For the second stage, data latch, we made the port B it uses to communicate with buffer passive as well as the port V it uses to communicate with the array (see Section II-C). Port V must be passive to make it possible for DMX(b, x) to initiate the parallel write. Latch's reshuffled bit-level HSE sequence reads (see Appendix part C)

```
# data latch (M) #
*[[bi];bo+;[vi];vo+;[˜bi];bo-;[˜vi];vo-].
```

V starts when B is halfway, hence, the latch provides a slack of half. This amount is clearly sufficient, since buffer only needs to see bo+, which corresponds to its [qi], to complete its C communication. The buffer can even get halfway through its next C communication, but it must wait for bo− to happen, which corresponds to its [qi], to proceed to completion. Thus, the buffer can complete a second C communication while V is only halfway—before even the write-request has been cleared (i.e., [˜vi]). Consequently, this reshuffling gives us a half-cycle more slack than we need.[5]

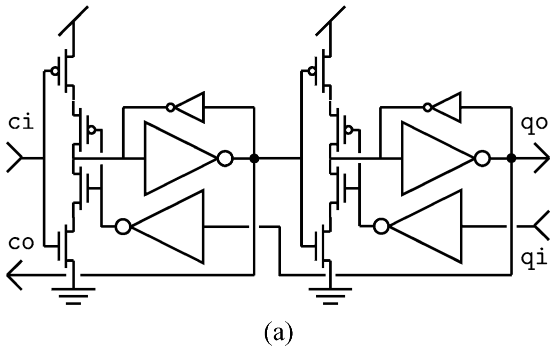
Compiling the latch's HSE into PRS (see Appendix part C) yielded the circuit shown in Fig. 11(b). bo goes high as soon as bi becomes high, provided vi is low. Now vi must become high before bo can drive vo high. When bi becomes low it cannot clear bo unless vo is high. When that happens, and bo is cleared, vi also must become low before vo is cleared. This sequencing ensures that data cannot be overwritten before it is read—even though clearing bo enables new data to be presented. Inactive cells require extra care, as explained in Appendix part C.

Finally, to implement ROW(x) [see Fig. 5(b)], we made its R and C ports both passive and made its P_k ports active (see Section II-C). The reshuffled bit-level HSE for the row reads (see Appendix part D)

```
# row cell (R) #
*[[ri&ci];po+;[pi];ro+;[˜ri&˜ci];po-;
[˜pi];ro-]
```

where we treat ci as data that arrives on R. Thus, there is no need for a separate acknowledge for C; ro acknowledges the latch as well as the decoder. Compiling this HSE into PRS (see

[5]The receiver will hang if the same column address appears thrice, because, while the first goes in the latch and the second is held in the buffer, there is nowhere to put the third; the address-latch provides less than a half-cycle.



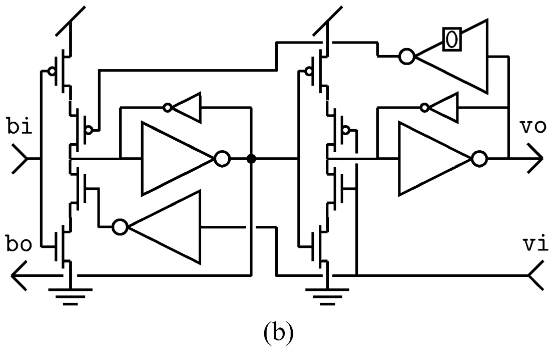


Fig. 11. Column buffer [P] and data latch [M]. (a) Interfaces column decoder (ci, co) with data latch (qo, qi). (b) Interfaces column buffer (bi, bo) with array (vo) and row decoder's C-element (vi). The zero-tag indicates that the inverter's output is forced low during reset; this clears the pipeline.

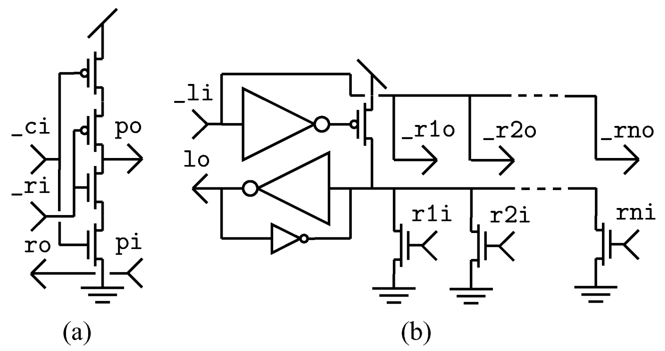


Fig. 12. Event-recipient interface [R] and acknowledge-OR. (a) Interfaces data latch (_ci) and row decoder (_ri, ro) with event-recipient (po, pi). (b) Broadcasts a request _li, to all n ports and ORs their acknowledges r1i, r2i, ... or rni, together to create a single one lo.

Appendix part D) yielded the circuit shown in Fig. 12(a)—a C-element (the staticizer was eliminated to save area). po goes high when _ri and _ci are both low, and it goes low when they are both high. Thus, the cell makes sure its row select and column data lines are both clear before clearing po, its request. Then, it simply copies pi, the acknowledge, to ro.

We combine ROW(x)'s bit-level acknowledges to generate a single acknowledge for that row using the staticized wired OR gate shown in Fig. 12(b). This gate produces an output when at least one cell acknowledges. Nevertheless, the remaining cells have sufficient time to read the column data, as the row-spanning wired OR line is slow due to its large capacitance. The OR gate's output, lo, is cleared when all the cells' acknowledges are clear. Until this is the case, _li cannot clear lo because the pFET is sized to be weaker than the nFETs.

IV. Summary and Conclusion

We have described an address-event receiver that writes all events destined for a particular row in parallel. While these events are being written, it decodes the next burst's column addresses, in preparation for the next parallel write. Bursts consist of a sequence of addresses: one for the row and additional ones for the column of each active cell in that row, plus a termination signal. They are communicated using a three-wire handshake: a row request, a column request, and a common acknowledge. In return for the extra request line, input pads are cut by 50% without sacrificing throughput as the row address is not repeated.

In terms of cell area, the cost of the parallel-write design is practically the same as prior receiver designs. Prior designs require a four-transistor NAND gate to combine the row- and column-select signals (this can be reduced to three if an NMOS-style design is used [22]), similar to our four-transistor C-element [see Fig. 12(a)]. They require an additional transistor to pull down the acknowledge line, similar to our staticized wired-OR [see Fig. 12(b)]. However, we made our design efficient by eliminating the staticizer, which would have added four more transistors. It is acceptable to cut corners here because the output is in the high-impedance state (i.e., `_ri` $\neq$ `_ci`) only briefly. Thus, the increased throughput and scalability parallelism offers [20] is attained at no 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 six asynchronous logic circuits that, together, can be used to implement a burst-mode word-serial address-event receiver of any desired size. These circuits include an improved decoder that eliminates glitches by using dual-rail encoding. 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 receivers for three generations of chips, fabricated in 0.6-, 0.4-, and 0.25-μm technology [20].

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` $\rightarrow$ `po+`, `~pi` $\rightarrow$ `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 $*[P; A]$, where P is passive and A is active, could be augmented with the state variable `s` as follows:

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

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

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

which we used for the decoder (see `#pipeline(E)#` in Section III-B and Fig. 10), 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 such sequences into PRS, multiple preceeding waits are ANDed together (e.g., `~ai&pi` $\rightarrow$ `po+`) and preceeding actions become guards too (e.g., `po` $\rightarrow$ `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` $\rightarrow$ `b+`) and pull-down (`d` $\rightarrow$ `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` $\rightarrow$ `c-`, `~b` $\rightarrow$ `c+`} requires a staticizer, since `a` = `~(~b)` is not an identity. However, {`a&b` $\rightarrow$ `c-`, `~a|~b` $\rightarrow$ `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. Demux

Making DMX's A port passive and its R and C ports active (see Section III-A) yielded these HSE sequences for the three-to-four-wire converter

```
*[[ari];ao+;ro+;[ri];[~ari];ao-;ro-;[~ri]]
||*[[aci];ao-;[~aci];ao+;co+;[ci];co-;[~ci]]
```

where two-phase handshakes communicate the row address and the end-of-burst signal (ϕ) on A and on R. To avoid storing the addresses locally, we moved the R communications into the middle of the A communications. This way, `ro+` occurred immediately after `[ari]`, passing on the address immediately. And `ao+` did not occur until `[ri]`, stalling till the address was read. Downward transitions follow the same sequence. We reshuffled the column-address communications in the same way, even though the second halves of these handshakes are meaningless. Thus, we obtained the sequences presented in Section III-A (`#demux(U)#`).

We compiled these reshuffled sequences into the following PRS:

```
ari->ro+    ari&~_aci->co+    ri&~ci->ao+
ari->ro-    ~ari|_aci->co-    ~ri|ci->ao-.
```

We strengthened the guard for co*scriptstyle*+ with ari, otherwise, it would fire at start-up, before we have the chance to set _aci high. We weakened the complementary guard to make the gate combinational. The corresponding circuit is shown in Fig. 8.

B. Delay

To delay delivering row addresses directed to DMX's R port to the row decoder until the burst ends (see Section III-A), our initial choice for an HSE sequence was

```
*[[gi];go+;[˜gi];go-;po+;[pi];po-;[˜pi]]
```

which has an active input port G and an active output port P. However, we postponed go− to avoid creating an ambiguous state, which would have required us to introduce a state variable. However, go− must occur before po−, otherwise, another ambiguous state would occur. We also postponed [˜pi] as long as we could to maximize the time we have to decode the address and write data to that row. Relocating this wait to just before po+ occurs in the next cycle gave us the reshuffling presented in Section III-A (#delay(D)#).

In compiling the reshuffled HSE, we introduced a local variable, u, that is cleared when gi becomes false and set when go becomes false. Hence, the sequence we implemented was

```
*[[gi];go+;[˜gi];u-;[˜pi];po+;[pi];go-;
u+;po-].
```

The following PRS resulted:

```
˜po&gi->go+    go&˜gi->u-    ˜u&˜u->po+
 po&pi->go-    gi|˜go->u+        u->po-.
```

We strengthened the guard of u− with go to disable it when u+ fired. And we weakened the guard of u+ to make the gate combinational. Notice that introducing u avoids a three-transistor chain in po's pull-up, which would compromise performance. The corresponding circuit is shown in Fig. 9.

C. Buffered Data Latch

Making LTH's C input port passive and its V output port also passive (see Section III-C) yielded the following bit-level HSE sequence:

```
*[[ci];co+;[˜pipi];co-;
[vi];vo+;[˜vi];vo-].
```

We introduced an intermediate communication in order to obtain more slack, and thereby expanded this sequence into two concurrent processes

```
*[[ci];co+;[˜ci];co-;qo+;[qi];qo-;[˜qi]]
*[[bi];bo+;[˜bi];bo-;[vi];vo+;[˜vi];vo-]
```

where the intermediate communication is called Q in the first process and called B in the second one (i.e., the active Q port is tied to the passive B port).

Reshuffling the first process gave us the HSE sequence presented in Section III-C (#buffer(P)#). Specifically, we advanced qo+ to the middle of C to eliminate the ambiguous state created by returning to the initial state halfway through the sequence. We restored symmetry by postponing [˜qi] to the next cycle, placing it immediately before qo+'s new location. Reshuffling the second process gave us the other HSE sequence presented in Section III-C(#data-latch(P)#). All we did was to postpone the second half of B to the middle of V.

We compiled the reshuffled buffer sequence into the following PRS:

```
˜qo&ci->co+    co&˜qi->qo+
qo&˜ci->co-    ˜co&qi->qo-.
```

The corresponding circuit is shown in Fig. 11(a).

We compiled the reshuffled data-latch sequence into the following PRS:

```
˜vi&bi->bo+    bo&vi->vo+
vo&˜bi->bo-   ˜bo&˜vi->vo-.
```

We guarded bo+ with ˜vi instead of ˜vo to prevent inactive cells from setting bo after vi becomes high. Otherwise, bo+ would fire when bi becomes true, since ˜vo is true for inactive cells. It would be followed by vo+, since vi also is true. Thus, the next burst's events would end up in the row that the current burst is being written to. Using the global signal, vi, instead of the local signal, vo, prevents this scenario, postponing bo+ in inactive cells until the on-going write is completed, at which point vi becomes false. The corresponding circuit is shown in Fig. 11(b).

However, blocking bo+ with vi may also lock out the last event in the current burst. Since the pipelined column decoder provides almost half-a-cycle of slack, the three-to-four-wire converter's C communication finishes while the buffer's C communication is only halfway through (see Fig. 6). That is, the converter issues ro− at the same time buffer issues qo+, which corresponds to data-latch's bi (see Section III-C). As ro− triggers the row-address-delay block to issue the write-request (i.e., vi above), correct operation requires sufficient delay to ensure bi becomes true first. Fortunately, this timing assumption is easily satisfied, as the buffer-to-data latch path involves just one channel, while the other path involves several, and includes signaling off-chip.

D. Row Cell

Making ROW's R and C ports passive and its P port active yielded this bit-level HSE sequence

```
*[[ri];ro+;[ci];co+;[˜ci];co-;
po+;[pi];po-;[˜pi];[˜ri];ro-]
```

where two-phase handshakes implement the pair of R communications (see ROW(x) in Section II-C). We chose to ran C in lock-step with R, which made `co` redundant. And we moved the first and second halves of P to the middle of the first and second R communications, respectively. Keeping `ro-` at the end guaranteed mutual exclusion. Hence, we obtained the sequence presented in Section III-C (`#row-cell(R)#`).

We compiled the reshuffled sequence into the following PRS:

```
  ri&ci->po+      pi->ro+
 ~ri&~ci->po-    ~pi->ro-.
```

The corresponding circuit is shown in Fig. 12(a).

ACKNOWLEDGMENT

The author would like to thank K. Zaghloul for reviewing the design and K. Hynna for invaluable help simulating and refining it. He would also like to thank one of the anonymous reviewers for his/her comments, which markedly improved the paper.

REFERENCES

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

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

[3] E. Culurciello, R. Etienne-Cummings, and K. A. Boahen, "A biomorphic digital image sensor," *IEEE J. Solid-State Circuits*, vol. 38, pp. 281–294, Feb. 2003.

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

[5] 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, May 1993.

[6] 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.

[7] C. M. Higgins and C. Koch, "Multi-chip motion processing," in *Proc. Conf. Advanced Research in VLSI*, vol. 20, 1999, pp. 309–322.

[8] W. Yang, "A wide-dynamic range low-power photosensor array," in *Proc. Int. Solid-State Circuits Conf.*, vol. 37, 1994, p. 230.

[9] 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.

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

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

[12] W. Maass and W. B. C. M., Eds., *Pulsed Neural Networks*. Boston, MA: MIT Press, 1999.

[13] K. A. Boahen, "A burst-mode word-serial address-event link—I: Transmitter design," *IEEE Trans. Circuits Syst. I* , vol. 51, pp. 1269–1280, July 2004.

[14] M. Sivilotti, "Wiring considerations in analog VLSI Systems, with application to field-programmable networks," Ph.D. dissertation, Dept. Comp. Sci, California Inst. of Technol., Pasadena, CA, 1991.

[15] 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.

[16] 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.

[17] J. G. Elias, "Artificial dendritic trees," *Neur. Comput.*, 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] 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.

[20] K. A. Boahen, "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.

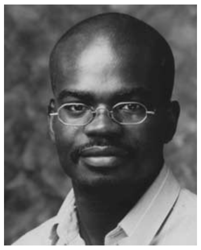
[21] A. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Proceedings of UT Year of Progamming Institute on Concurrent Programming*. Reading, MA: Addison-Wesley, 1990, pp. 1–64.

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

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

[24] ——, "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.

[25] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, 1989.



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.