Four State Asynchronous Architectures

Anthony J. McAuley

Abstract— **This paper describes a new approach to high-performance asynchronous architectures offering significant advantages over conventional clocked systems, without some of the drawbacks normally associated with asynchronous techniques. As the level of integration increases, an asynchronous wavefront array designed using the techniques described will have three important advantages over the equivalent synchronous systolic array: faster throughput (rate at which data are clocked through a system), reduced design complexity, and greater reliability. While the latter advantage is well known [1], the first two, which are a consequence of this new approach, make self-timed systems much more attractive for high-performance applications. The cost of our proposed asynchronous architecture is silicon area, which will be at least double that of the equivalent synchronous design. The benefits and drawbacks of using our asynchronous technique are highlighted using three wavefront arrays: two one-dimensional multipliers and a two-dimensional sorter. All three can be built using just one basic building block, and simulations in 2 μm CMOS show they are capable of throughput of 250 MHz for static logic implementation and over 400 MHz for a dynamic logic implementation.**

Index Terms— **High-speed logic design, multiplication, self-timed system, sorting, VLSI, wavefront array.**

I. Introduction

THIS paper explains how a four-state asynchronous wavefront array is able to exploit the full switching speed potential of submicron technology while maintaining small design complexity.

In the VLSI era, using small, simple, and repetitive building blocks have contributed to keeping complexity under control. A good example of this is the bit-level systolic array [2]–[4] with one cell repeated many times each performing the same simple operation. Though these systems are capable of fast parallel computations, they suffer a number of drawbacks associated with the need to drive hundreds, or even thousands, of cells with a single global clock.

The clock speed must be conservative enough to allow for worst case logic delay and must include the delay in the clock drivers themselves. Clock synchronization between cells and between different phases of the clock is difficult to maintain at high speeds and can become a source of transient errors [1]. Having all cells in lock-step requires power to be concentrated at the clock edge, slowing switching speeds, and accentuating the metal migration problem. Moreover, the design of the clock drivers and clock distribution is difficult and is not directly scalable.

Manuscript received May 20, 1989; revised October 12, 1990.

The author is with the Network Services Research Laboratory, Bellcore, Morristown, NJ 07962.

IEEE Log Number 9103110.

A wavefront array [5], [6] is broadly equivalent to a systolic array, except that individual cells are data driven and require no global clock. Instead of having a master controller choreographing each cell (via the clock line), a wavefront array cell detects when its inputs are valid to change its outputs. This distributed choreography is achieved using a variety of handshaking protocols [7]–[13].

This paper describes a modified approach to wavefront arrays based on the compiling techniques of Martin [11], but uses only one basic building block, for lower design complexity, and uses a more efficient communication code and logic implementation for higher throughput. The principal advantages over both synchronous and traditional asynchronous architectures are increased speed and reduced complexity; it also offers excellent reliability with no more area than some of the best traditional asynchronous approaches.

A. Existing Techniques

Those unfamiliar with synchronous and asynchronous communication protocols should first read a good introduction to self-timed systems: such as Seitz [1]; those familiar with the field may skip Sections II and III.

This paper concentrates on bit-level implementations, because of their application to low complexity systolic/wavefront arrays. Section II gives an overview of the differences between the synchronous and asynchronous techniques, showing that, as the level of integration increases, asynchronous communication becomes increasingly attractive for applications where area is not critical.

Section III overviews two general techniques for communicating asynchronously. When the handshaking and data logic are separated, we call this "single-rail" logic; and when the handshaking and data logic are integrated, we call this "double-rail" logic.

B. New Approach to Asynchronous Communication

The core of our approach to asynchronous logic is four-state coding using "double-rail" logic. Section IV describes how data and handshaking information are coded using the traditional three-state coding [1], [11] and our proposed four-state coding. To illustrate how the latter works, Section V describes the design of a simple asynchronous shift register.

The basic cell used in the shift register is an asynchronous version of the clocked master slave latch, in which the input is passed to the output after handshaking. Section VI describes how to design a more general cell using four-state coding with inputs from many cells mapped into an output, which may also be received by more than one cell. The design of an asynchronous 2:1 multiplexer concludes this section. Section VII gives examples of three low complexity designs using just this multiplexer: two multipliers and a sorter. Section VIII brings in some practical insights from a test chip fabricated using four-state coding and dynamic logic.

II. A Comparison of Bit-Level Arrays

Kung [2] has shown that for high performance and low complexity it is best to have a network of (identical) simple cells that pass data in a regular fashion. These bit-level array architectures are called systolic arrays when they are synchronous and wavefront arrays when they are asynchronous.

The main characteristic of the clocked (synchronous) systolic arrays is that each cell carries out its computations at the same time as every other cell. For bit level structures there are potentially thousands of cells relying on a single global clock to tell them when valid data are available and when it can change its output. To ensure the correct inputs are at the right place at the right time, the distributed clock must be kept well synchronized throughout the array.

With clockless (self-timed, or asynchronous) wavefront arrays, data are pumped around in the same direction; but the timing is controlled by the elements themselves, not by a common clock. To keep computations ordered, the asynchronous elements must know when data are valid and when they can change their output. It is our belief that these wavefront arrays are inherently less complex to design, faster, and more reliable.

The asymptotic characteristics of the synchronous and asynchronous techniques are described below. We summarize these using four key VLSI design factors: design complexity, speed, area, and reliability. Section VIII also describes some practical differences. A more complete discussion of timing problems is given by Seitz [1].

A. Design Complexity

A key lesson VLSI designers learned from software designers is to divide a problem into modules that can be designed separately. To reduce complexity, it is necessary for the boundary between modules to be well defined and simple. An important boundary condition is to know when the data communicated are valid.

Synchronous systems force all data to be valid at the edge of a global clock. The designer must allow for clock skew, either between cells or between different phases of the clock. Also, each element has fixed setup and hold times. Overcoming these timing problems is far from trivial and is frequently the cause of devices being either slow, unreliable, or not working at all [1]. Furthermore, the clock driving circuit and clock distribution network (e.g., the number and size of the various stages in the clock buffer) do not scale linearly with the main array. As technology allows bigger and bigger arrays, maintaining synchronization, while trying to achieve maximum performance, will grow significantly more complex [5].

Asynchronous designs have no global clock, so each cell keeps time to itself; however, they have complexities of their own. First, races and hazards [14] need more careful consideration. Second, the logic to detect when data are valid requires extra circuit design.

We show that the asynchronous complexity problems can be overcome using a standard cell approach.

B. Speed

Though not always appreciated, the global clock can significantly limit the performance in a large system. Even today, the logic delay in a bit-level systolic element is typically half that of the "safe" clock frequency. There are several reasons for this difference:

1) The large load on the clock buffer, driving thousands of gates, causes significant delay between the chip clock input and the buffered clock signal.
2) Propagation delay down distribution lines means that cells furthest away from the buffer receive the clock later than those nearest the buffer. As technology advances, the clock distribution time becomes a more significant fraction of the switching time.
3) Part of the clock period must be set aside to allow for skew.
4) The clock speed must be a conservative worst case, both in terms of fabrication and environmental parameters, if the chip is to operate reliably.
5) Cells switch virtually simultaneously, causing the power supply lead inductance to become a more significant limitation on switching speed.

Some of these problems can be minimized. For example, using many power pins and wide power tracks reduces the effect of the current spike at clock edge. But, as technology advances, allowing more and faster cells to be fabricated onto a single chip, synchronous inefficiencies will become increasingly significant. Reducing the minimum feature size does not affect the diffusion delay per minimum feature size [2]; but, if the total area remains constant, the delay between the clock at the driver output and cell input increase. As the number of cells (n) increases, this speed reduction is approximately proportional to log n; where this logarithmic factor is observed principally in the increased clock buffer delay.

Asynchronous systems do not need a global clock and different cells switch with a more even distribution over time; therefore, provided data communication remains localized, the delay is independent of the number of cells used. Traditionally, the delay in an asynchronous cell is significantly greater than the equivalent synchronous cell, because it performs the function and detects when the state should change. Also, it is more difficult to exploit the speed advantage of dynamic logic [14], because we cannot guarantee minimum state refresh.

We will show that asynchronous logic can be made faster using four-state codes and more efficiently designed detection logic. Also Section VIII describes the use of dynamic logic to improve basic switching speed. The combination of these techniques makes asynchronous logic faster than the equivalent synchronous designs for either large arrays or arrays built using fast switching elements.

C. Area

In a synchronous systolic array, all cells can share the same clock generation and buffer circuits; so cell design is kept simple, since they can assume data are valid on the clock edge. As the number of cells gets larger, the size of the final clock buffer stage grows proportionately. For n cells the area of the clock buffer circuitry grows at a rate of about $n/3$, assuming a $1:3$ step up ratio between the driver and the load (see section 7.5 of reference [1]).

Every asynchronous wavefront array cell requires its own circuitry to detect when data are valid; therefore, as the number of cells increases, the total area of this detection logic will grow at the same rate. For n cells, the area of the detection circuitry is directly proportional to n.

The inherent area advantage of synchronous designs cannot be overcome; however, both grow at the same rate (linearly) with increased system size. Furthermore, we show that the area disadvantage of wavefront arrays can be kept within reasonable limits, even for bit-level cells, by careful logic design.

D. *Reliability*

The clock in a synchronous circuit can be a source of transient and permanent errors [1]. Even when modules communicate correctly under ideal or typical conditions, timing problems can still arise. Changes in speeds, caused by processing or the environment, can make the system fail even if we chose a conservative clock speed. For example, it could exaggerate clock skew and require increased setup and hold times. For systems running at their maximum clock frequency, this means reduced reliability.

In synchronous schemes, where all elements switch at the same time, there is a large current spike just after the clock edge: particularly for a bit-level systolic designs where each bit has its own latch. This current spike makes the chip more susceptible to metal migration which is dependent on peak current.

Another, less well known problem associated with the clock occurs in circuits designed with fault tolerance. If the clock lines are left nonswitchable a short in the clock line will completely kill the device, independent of any data switching mechanisms. Fault tolerant circuits require bad elements to be switched out [15]; but, it is not desirable to switch lines carrying a large current, such as the clock line, since it is costly in area and significantly degrades performance.

Asynchronous arrays do not have the robustness problems associated with a clock. However, the design must be done carefully to avoid races and hazards. Also, the detection circuitry requires more average power than a clock driver, though its requirements are more distributed over time. Furthermore, because of their larger area, an asynchronous array is more subject to soft errors caused by radiation and hard errors caused by processing defects.

We believe that asynchronous circuits have an inherent reliability advantage, provided they are well designed using compiler like techniques, such as those originally proposed by Martin [11] and adapted here.

III. Asynchronous Logic

Transferring one bit of binary information (i.e., a "1" or "0") asynchronously without a global clock can be done in a variety of ways [1]. A one way control scheme transmits data between cells, together with information telling the receiver when new data are on its way. The receiving cell must process the data as it arrives or store it in an elastic buffer. For bit-level wavefront arrays, it is too costly to store data inside every cell; therefore, with the possible exception of the chip interface cells, some form of handshaking is preferable.

This section briefly describes two classes of handshaking. Our distinction is based on whether the handshaking logic is integrated with the data logic: "single-rail" logic keeps data separate, "double-rail" logic combines it.

A. *"Single-Rail" Logic Handshaking*

"Single-rail" logic has two separate, almost independent, pieces of logic: the data logic and the handshaking logic. The handshake logic communicates with the data logic through the local "clock" (each cell has its own independent clock, whose definition is strictly local). The clock can be stopped synchronously and started asynchronously (see section 7.8.4 of [1] for more detail). The data logic is designed in exactly the same way as for a synchronous system, except for the clock source. Input latches store the incoming data and combinational logic maps this latched data into the cell's output data. Because each binary variable is represented by a single wire, we call this "single-rail" logic.

The handshaking logic generates the clock for the input latches using a local clock and a handshake protocol with cells that transmit and receive its data. The derived clock is used to latch data in, knowing that the input data are valid and the previous output is no longer required. This technique assumes that data are not significantly delayed between cells, relative to the handshaking signal. If it were, the receiver may sample the data at the wrong time. Fortunately, with current technology this race condition is not a problem inside the chip; especially when data are only communicated to nearest neighbors.

"Single-rail" logic handshaking was first used in wavefront arrays by Kung. The scheme [7], [8] works well; however, it is slower than the equivalent synchronous system, since it requires over two local clock periods to latch the incoming data.

B. *"Double-Rail" Logic Handshaking*

"Double-rail" logic integrates the handshaking and data logic, eliminating all clocks. To define when data are sampled, it is necessary to use more than two states to represent a single bit of information. For binary logic cells, each input and output requires a pair of wires containing both the data and the handshaking information; hence, we call this "double-rail" logic. The handshaking information, contained within the wire pairs, disables output changes until two conditions are met:

1) The previous output has been used by all cells that receive it.
2) All the new inputs are valid.

The next section describes two codes used to represent the data and handshaking information.

IV. "Double-Rail" Codes

This section describes three-state [11] and four-state coding of data and handshaking information. Even though four-state requires an extra state, it is faster.

A. Three-State Coding

Three-state coding has been used in a number of designs [7], [16], where the code has three detectable states: logical 1, logical 0, and a null state N. After transmitting each bit of data (1 or 0), a null (N) is transmitted as a separator. Data are defined as the value between two successive nulls, and a null is defined as what is in-between two bits of data: in both cases without reference to time.

Data are a wave of ternary signals of varying duration with every odd signal being data and every even signal a null. It is possible to implement ternary logic directly, using say +Vdd, Vss, and −Vdd to represent the three levels. The resulting cells, however, are larger and more complex than their equivalent binary representation; so ternary signals (1, 0, and N) are coded using two binary digits. One representation [7] is

$$\begin{aligned}00 &= \text{acknowledge—null } (N),\\ 01 &= \text{data value—logical 0 (0)},\\ 10 &= \text{data value—logical 1 (1)},\\ 11 &= \text{Not allowed.}\end{aligned}$$

This particular coding prevents both wires from switching at the same time: so the output goes through a Gray scale [17]. Fig. 1(a) illustrates how data are mapped into this two-bit, three-state code and shows that only one bit changes in each transition—preventing hazardous operation.

The use of nulls to define data simplifies the detecting logic, but is inefficient. It limits throughput because two signals (data and a null) must be propagated serially for every bit of useful information transmitted. Thus, in the example shown in Fig. 1(c), three-state coding requires four times as many bits to pass the same information as a synchronous data stream.

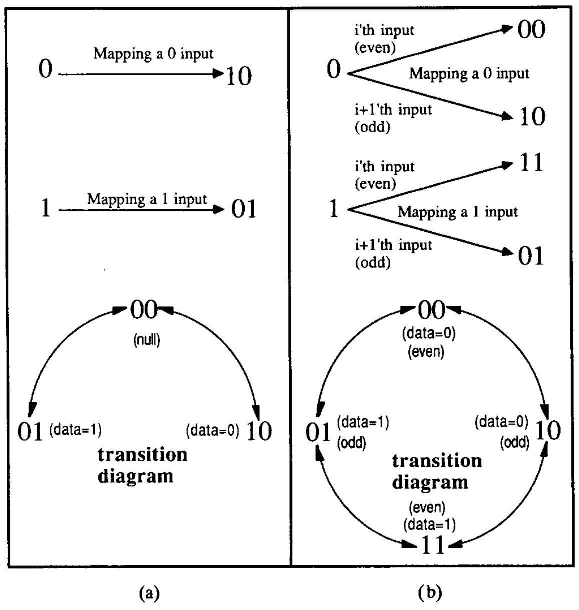
B. Four-State Coding

The four-state coding scheme proposed here allows the data wave to be in one of four states: two representing logical 1 ($P1$ and $Q1$) and two representing logical 0 ($P0$ and $Q0$): where P and Q can be thought of as two phase representations. To transmit data, we alternate between the P and Q phases. Because there is always a transition, a null is no longer needed to delimit the data. For example, the coding of the series $0, 1, 1, 0, 0, 1$ would be $P0$, $Q1$, $P1$, $Q0$, $P0$, $Q1$.

Four-state coding ($P1$, $Q1$, $P0$, and $Q0$) can be coded in binary logic using two digits. One such representation is

$$\begin{aligned}00 &= \text{data value}: \text{ logical 0 } (P0)\\ 01 &= \text{data value}: \text{ logical 1 } (Q1)\\ 10 &= \text{data value}: \text{ logical 0 } (Q0)\\ 11 &= \text{data value}: \text{ logical 1 } (P1).\end{aligned}$$

With this coding, the left bit represents the parity (p) and the right bit the data (d). The logical exclusive-OR (XOR) of p and d represents the even (P) and odd (Q) phases.



Data stream	0	1	1	0	0	1
3-state coding (a)	00 10	00 01	00 01	00 10	00 10	00 01
4-state coding (b)	00	01	11	10	00	01

(c)

Fig. 1. Double rail code, with three-state and four-state coding (a) Three-state. (b) Four-state. (c) Example.

Alternate between the P and Q phase representations, we guarantee to change exactly one bit for every transition—so the output is again Gray scale: that is, either p or d changes, but not both. Every even bit transmitted is 00 for "0" and 11 for "1"; while every odd bit transmitted is 10 for "0" and 01 for "1".

Fig. 1(b) summarizes the mapping and transitional information for four-state coding. By comparing the example, shown if Fig. 1(c), of coding in both three-state and four-state, it can be seen that four-state requires half the number of bits to communicate the same information. Four-state code also reduces the area required to implement a cell, because it does not store null values.

The wave is thus a series of four level signals of variable duration defined without reference to time. An odd segment is defined as what is in-between two even segments and an even segment is defined as what is in-between two odd segments of data.

V. Four-State Asynchronous Shift Register

To illustrate the four-state asynchronous coding scheme we shall describe a simple example. The hardware is based on the Muller C element [1], [6], [11] with four-state "double-rail" logic handshaking. The example is analogous to a clocked shift register based on a master–slave latch. Because it is asynchronous, however, the input and the output are independent: so it functions as a first-in-first-out (FIFO) buffer. This section describes the operation using a simplified timing model. A simulation, based on a real CMOS transistor layout, is described in the Appendix.

A. The C11 Cell

C11 is our four-state finite state machine corresponding to a clocked master slave latch. Fig. 2 shows its block diagram and Table I shows its set–reset state table. The two inputs $(A[i-1], A[i+1])$ and one output $(A[i])$ each have a pair of wires coded to represent the four-state double-rail code.

Fig. 4 shows the C11 cells tied together in a chain, like a clocked shift register. In this arrangement, $A[i]$ is the output $(ad[i])$ and $(ap[i])$ of the cell itself, $A[i-1]$ is the output $(ad[i-1])$ and $(ap[i-1])$ from the previous cell to the left, and $A[i+1]$ is the output $(ad[i+1])$ and $(ap[i+1])$ from the next cell to the right.

Of the 16 possible input states, half change the output state. The truth table shows that each cell only accepts odd phase data when the previous (left) cell has odd data and the next (right) cell has even phase data. Also, the cell only accepts even phase data when the previous cell has even phase data and the next cell has odd phase data. Each cell in the chain alternatives between odd and even phases, as shown in the asynchronous shift register description below.

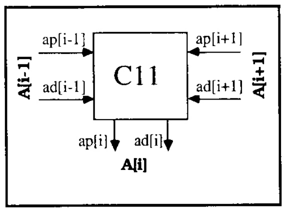


Fig. 2. C11 block diagram.

TABLE I
C11 State Table

A[i-1] (ap,ad)	A[i+1] (ap,ad)	A[i] (ap,ad)
00	01	00
00	10	00
01	00	01
01	11	01
10	00	10
10	11	10
11	01	11
11	10	11
else		No change

B. An Asynchronous Shift Register

Figs. 4, 5, and 6 shows 18 "time frames" of the four-stage asynchronous shift register. In the description that follows, each time frame looks like a separate clock period. Though this simplifies the description, it is important to realize that events in different cells occur at their own pace and are not synchronized.

In the first time frame (equivalent to the first clock pulse), shown at the top of Fig. 4, all C11's are reset to the 00 state and the input from their left and right are also 00. Table I shows that this is a stable state, since every C11 outputs a 00 when both inputs are 00.

Fig. 3 shows the input waveform applied to the first (leftmost) C11, representing a 1001100 information stream, and the outputs of various C11 cells which result from this waveform. Each frame number, labeled at the bottom of Fig. 3, corresponds to a time frame (**F**) in Figs. 4, 5, or 6. Fig. 3 and Figs. 4–6 are two different representations of the same information.

The first input is a logical 1; since the first phase is odd a 01 is input to the first cell ($F = 2$). This first C11 now has the input conditions shown in row three of Table I, so its output changes to 01 ($F = 3$). This is not a stable state as the second C11 now has an input corresponding to row three of Table I; so its output will go to 01 ($F = 4$). At the "same time," the input to the first cell can now be changed, since we know the first data have been latched: that is, the first cell has odd phase (01) on its output. The next input is a 0, so a 00 (the second phase is even) is input to the first cell ($F = 4$).

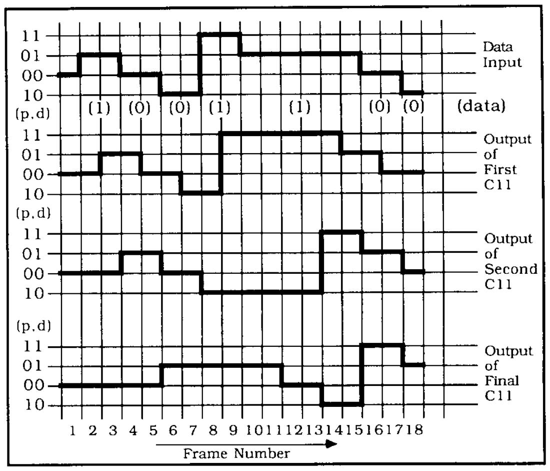


Fig. 3. Asynchronous FIFO timing diagram.

The effect of the first input change ripples through to the third C11 cell, which changes state like the first two ($F = 5$). At the "same time" the first C11 has inputs corresponding to the first row of Table I, so its output changes to 00 ($F = 5$). In time frame 6 ($F = 6$), the 01 from the third cell propagates to the fourth C11 at the end of the shift register. Also, the second cell has inputs corresponding to the first row of Table I, so its output changes to 00. Meanwhile, because we know the leftmost C11 has accepted the second data input (its output is even phase), the next input, a 0 (see Fig. 3) is loaded; so a 10 (the third phase is odd) is input to the first cell ($F = 6$).

Moving on to time frame 7 ($F = 7$), at the top of Fig. 5, the third cell and first cells change their inputs corresponding to rows one and five of Table I, respectively. Note that now the fourth C11 cannot change state unless the input from the right-hand side (00) changes: that is, until its output has been acknowledged the fourth C11 maintains a 01. Also,

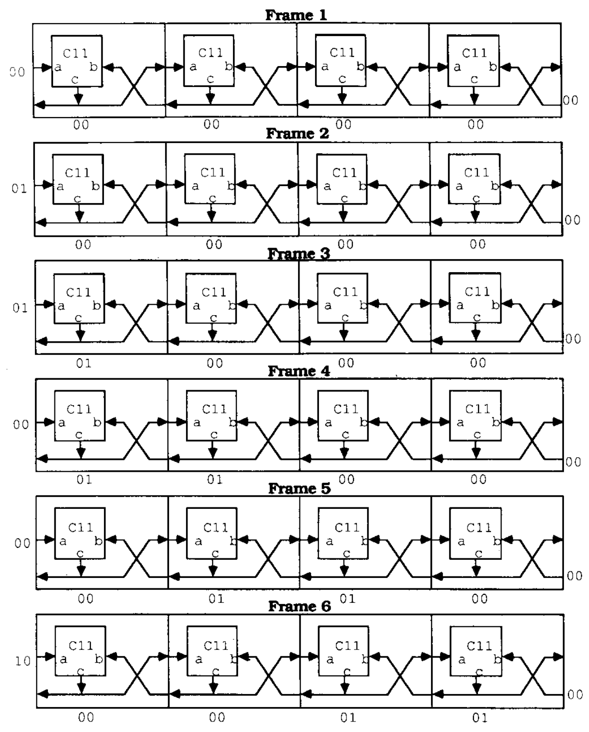


Fig. 4. First six "time frames" for an asynchronous FIFO.

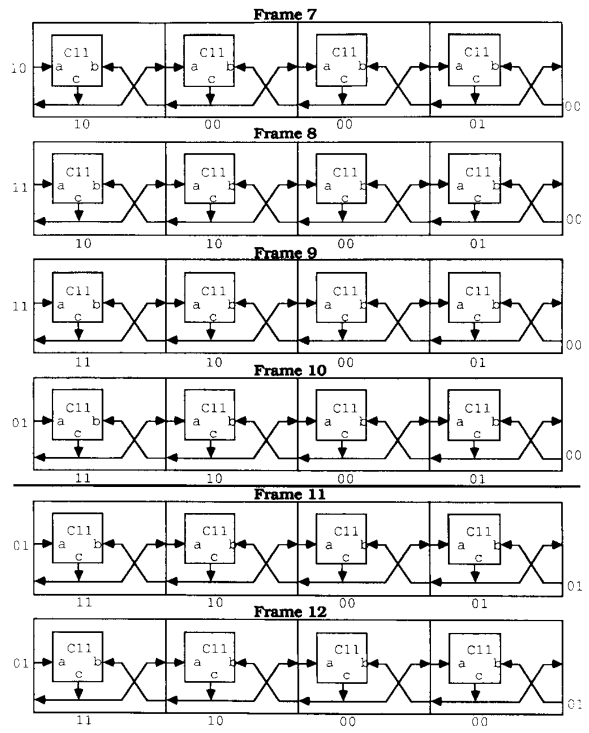


Fig. 5. Second six "time frames" for an asynchronous FIFO.

because the third cell does not receive an (even) acknowledge from the fourth cell, it will keep a 00 on its output. This blocking continues back down the chain, until in time frame 10 ($F = 10$) we are again in a stable position with the data values $1, 0, 0, 1$ stored in the four cells and a 1 waiting to come into the chain. No more data can be loaded into the shift register until we start strobing the right hand input.

In time frames 11–15, the data are removed. Thus, we acknowledge the first output by putting odd phase data (01) on the right-hand input ($F = 11$). We acknowledge the second input by putting even phase data (00) on the right-hand input ($F = 13$). In time frame 15 ($F = 15$) we acknowledge the third input with an odd phase (01). In the last three time frames, we both load data into the left-hand side and read it out of the right-hand side.

The four C11 cells operate as a elastic storage medium, with data loaded (if there is room) and removed (if there is data available), independently.

VI. General Purpose Asynchronous Cells

Every cell in the asynchronous shift register alternated between even and odd phased data. To keep the even and odd phases separate, a cell changes state only when the two surrounding cells were of opposite phase; ensuring that the old data were read and there is new data to read. The conditions necessary for successful handshaking can be summarized by rules a1 and b1, or alternatively by rule c1:

a1) The downstream receiving cell must have latched the data.

b1) The upstream transmitting cell must have new data.

c1) The upstream and downstream cells must have opposite phase.

Limiting the cell to just one input and driving just one output is clearly too restrictive. More generally, there are inputs from many cells and the output goes to many cells. This output is defined by a mapping of the inputs, while keeping the same phase as the inputs. The restrictions necessary to ensure the data are kept separated can be generalized from the three rules above:

a2) All the downstream receiving cells must have latched the data.

b2) All the upstream transmitting cells must have new data.

c2) All the upstream cells must be in one phase and all the downstream cells must have the opposite phase.

To design our basic wavefront array cell, we use just one type of cell—the multiplexer cell (MC). No separate combinational logic gates used; instead, even the simplest gate is built from a sequential MC. This both reduces complexity (only one cell to design) and improves throughput (increasing the number of pipeline latches). The design technique is analogous to designing sequential logic using just 2 : 1 multiplexers, with a master–slave latch on each output.

A. Acknowledge Combination

A transmitting cell does not need to know both the downstream cell's data and parity; all it requires is one bit indicating its phase. Thus, if a cell transmits to just one receiver, it

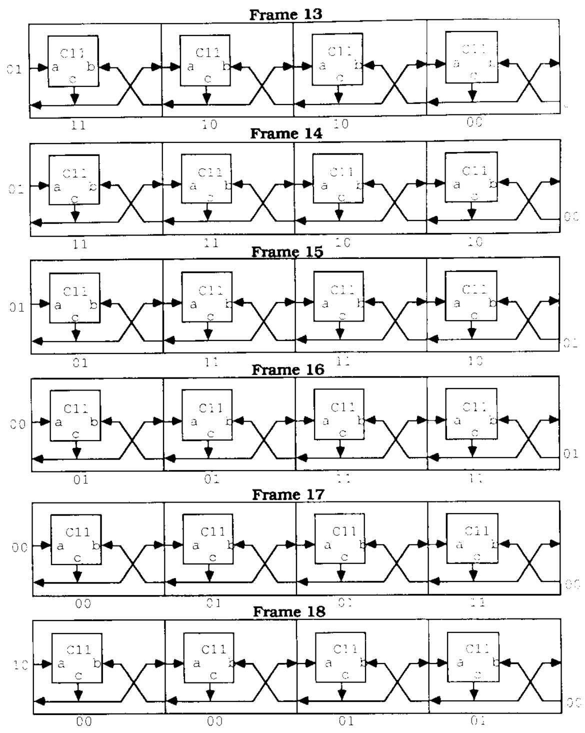


Fig. 6. Final six "time frames" for an asynchronous FIFO.

requires only the receiving cell's phase: e, which we say is low for even phases (00 and 11) and high for odd phases (01 and 10). Thus, e is the exclusive-OR (XOR) of p and d.

A cell broadcasting to multiple receivers must ensure that all receiving cells have processed the information by waiting for all the receiving cells to be in the same phase. Phase information from individual cells is combined in a single latch, which is set if all the phases are high and is reset if all the phases are low.

B. Mapping the Inputs

Fig. 7 shows the most general type of an asynchronous cell, $CxyZ$. It has x pairs (p and d) of inputs (labeled $A-I$), y 1-bit (e) feedbacks (labeled $R-W$), and an output pair (labeled M) which does the mapping Z on the inputs when the handshaking is complete. The state only changes when all inputs have the same phase and all acknowledges have the opposite phase; therefore, the more inputs there are, the sparser the set–reset state table becomes. For example, Table II shows the truth table for a C22F with two inputs (A, B) and two outputs (R, S) performing the function $F(A, B)$.

The truth table must be modified if the number of transmitters and receivers, or the function changes. Since the design of these asynchronous cells is nontrivial, we opt instead to design just one type of cell and build all others from this basic building block.

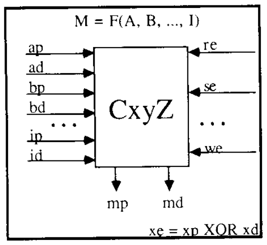


Fig. 7. $CxyZ$ block diagram.

TABLE II
C22 STATE TABLE

A (ap,ad)	B (ap,ad)	R (re)	S (se)	M (mp,md)
00	00	1	1	F(0,0)
00	11	1	1	F(0,1)
01	01	0	0	F(1,1)
01	10	0	0	F(1,0)
10	01	0	0	F(0,1)
10	10	0	0	F(0,0)
11	00	1	1	F(1,0)
11	11	1	1	F(1,1)
else				No change

F(i,j) = Output mappingwith ad=i & bd=j

C. The Multiplexer Cell

Fig. 8 shows the asynchronous 2 : 1 multiplexer cell (C32M or MC), with three data inputs (of two bits each) A, B, and C, two acknowledge inputs R and S (of one bits each), and an output M (of two bits). When the inputs and acknowledge obey the restrictions necessary for successful handshaking, the output mapping (M) is defined by the normal multiplexer function:

$$M = (A \text{ AND } C') \quad \text{OR} \quad (B \text{ AND } C)$$

where C is the select line. This MC is the key to our simplified, low complexity approach to asynchronous designs.

With combinatorial logic, it is possible to design any function efficiently using a tree of 2 : 1 multiplexers. We will use the same principle for asynchronous logic by building all function from MC's. Simple circuits, such as a pass mapping $(M = A)$, an OR mapping $(M = A \text{ OR } B)$, or an exclusive-OR mapping $(M = A \text{ OR } B)$, can be built from one MC. More complex functions can build using a tree of MC's. Section IV-D discusses the design of these trees.

Using a multiplexer to build a pass gate (i.e., for an asynchronous master slave latch) is somewhat redundant. However, the complexity reduction makes up for the extra serial transistor in the serial path (it does not change the number of logic levels). It is also redundant to use two acknowledges where only one is required; but, again, the complexity reduction was deemed more important.

A further reason for not building a large library of asynchronous gates (e.g., AND, OR, and exclusive-OR) is that four-state asynchronous logic is much less amenable to minimization. It is not possible to combine sum-of-product terms, since every term which changes the state is "surrounded" (in the Karnaugh map sense) by terms which do not change the state. Furthermore, many alternative techniques to minimize

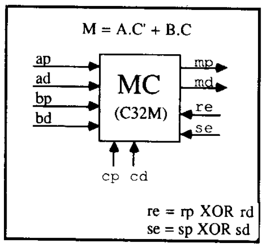


Fig. 8. Multiplexer (C31M).

the number of transistors, using exclusive-OR gates for example, produce critical races. Consequently, the area/transistor difference between a customized AND cell and an AND cell built from an MC is small.

1) The MC Truth Table: Table III shows the truth table for the multiplexer. The handshaking makes our four-state implementation significantly more complex than that for the equivalent synchronous multiplexer. The truth table may be implemented in different ways. However, before considering alternative logical implementations we must ensure the sequential logic is free from races and hazards.

2) Hazards and Races in Asynchronous Cells: Hazards and races are a well known problem in asynchronous circuit design [17]. We have already encountered this problem when choosing the binary representations for our four-state coding (to ensure Gray scale transitions). Here we look at another possible source of races: the way the logic is implemented.

Consider just one cell in isolation. Assume the inputs are connected through combinational logic to the set and reset inputs of a latch. If the rest of the circuit is designed correctly, then an input variable changes at most once for every change in state and there is only one possible change of state for any allowable change in variables. Even when these conditions are met, however, there is still a possibility of a race if the combinational logic is poorly designed.

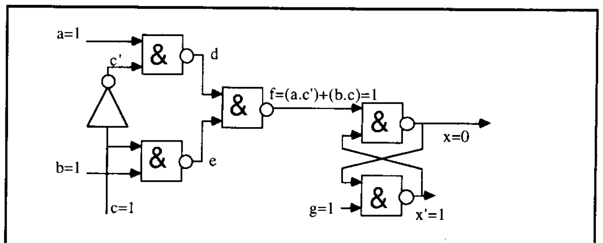
Take as an example part of a possible asynchronous multiplexer cell shown in Fig. 9. Driving c low (with all inputs initially high), there should logically be no change in the output (either at f or x). But, because of finite gate delays, e goes high before d goes low. The resulting low spike on f may, if it lasts long enough, set x high causing the circuit to malfunction.

In the above example, the hazardous race was possible because there existed two paths from the c input to the latch set input (f). To ensure no races of this kind it is sufficient [17] to ensure that only one path exists from any input to the set or reset inputs of the latches. This is the technique we adopted.

3) Logic Implementation: The best method of implementing the MC truth table of Table III without races (i.e., without multiple paths) depends on the technology and the requirements (e.g., is speed, area, or reliability the critical factor). Because of the area overhead associated with handshaking we chose CMOS technology which permits a high level of integration. For our purposes, we decided that the primary requirement for the MC was reliability, followed by speed, then area, and finally complexity. The latter being, for once, at the bottom of the list because the cell contains only a handful of transistors which are replicated over and over again.

TABLE III
MULTIPLEXER STATE TABLE

A (ap,ad)	B (bp,bd)	C (cp,cd)	R (re)	S (se)	M (mp,md)
00	00	00	1	1	00
00	00	11	1	1	00
00	11	00	1	1	00
00	11	11	1	1	11
11	00	00	1	1	11
11	00	11	1	1	00
11	11	00	1	1	11
11	11	11	1	1	11
10	10	10	0	0	10
10	10	01	0	0	10
10	01	10	0	0	10
10	01	01	0	0	01
01	10	10	0	0	01
01	10	01	0	0	10
01	01	10	0	0	01
01	01	01	0	0	01
else					No change



If c goes low, then for a short time there will be a low spike on f. If this lasts long enough, the circuit would malfunction and set x high.

Fig. 9. Example of hazardous asynchronous circuit.

We designed a dozen different variants of the MC cell with static and dynamic logic. Some expanded the number of states and others used carefully considered architectural assumptions to simplify the logic. Two implementations were particularly good:

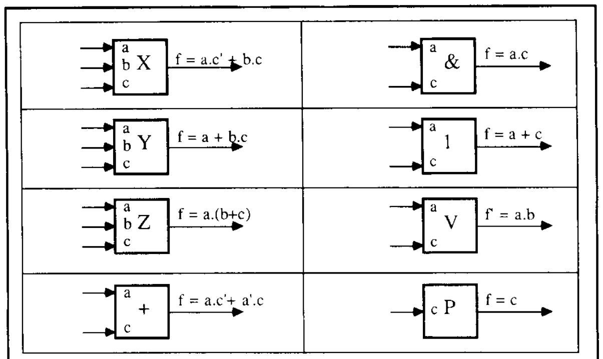
1) A fast dynamic version, dubbed MC^,
2) A reliable static version, dubbed MC*.

MC^ and MC* use full CMOS logic swings. They have only one logic level between data input and data output, and two levels of logic between their data inputs and the acknowledge signal: where each logic level has at most three transistors in series. Both MC^ and MC* differ from the truth table shown in Table III; however, for the purposes of this paper, we can assume MC^ and MC* are identical to MC.

D. MC-Graphs

Design of sequential logic using only MC's is analogous to designing in combinatorial logic using only multiplexers.

1) Simple Functions: Like normal multiplexers some input lines are tied to 0 or 1 to build simpler functions (such as an AND gate). Fig. 10 shows the symbolic form of some simple "gates" that we can make from an MC. How these functions are derived by tying down certain inputs to the power rail is omitted for simplicity. However, one key to their design is the way MC^ and MC* were built. Both MC^ and MC* allow the data or phase of the input or output to be inverted, without any cost in area, delay or complexity.



Note 1: Data or parity can be inverted without extra logic or delay. Data inversion is part in the function definition & parity inversion is represented by a small circle (o).
Note 2: The parity can be ignored - which is represented by a star (*).

Fig. 10. Examples of asynchronous mappings built from one MC.

2) Complex Functions: We make some simplifications to our architecture diagrams. First, we only use one wire to represent the pair of wires used in our four-state coding; second, we do not explicitly draw in the acknowledge lines. Since the acknowledge lines can be drawn in only one way after we have fixed the MC's, there is no information gained by explicitly drawing them. The result of these simplifications is an MC-graph, like those in Fig. 13, showing how each MC are connected together and what mapping is done (each box has a label to describing the function, such as a NOR gate from Fig. 10). How to design the MC-graph is best shown through examples.

VII. Four-State Asynchronous Architectures

This section describes the design of MC-graphs through three bit-level wavefront arrays: two multipliers and a sorter. Layout is simpler than for synchronous standard cell designs because we have only one basic building block and no clock distribution problems.

If the cells in the MC-graph are connected in a simple tree, with no loops, data can flow through without hindrance at the maximum rate allowed by the technology (assuming data are not blocked at the leaf nodes). Here, the delay between an MC receiving data and being able to receive the next bit of data is three gate delays (i.e., three combinational logic levels). This can best be seen in the shift register example, where a cell receiving data must wait until the data have propagated through itself (1 gate), the next cell (1 gate), and wait for the acknowledge from the receiving cell (1 gate). Thus, the average time between successive bits of data is three gate delays. With a 1 ns delay per gate, this is equivalent to a static synchronous system running at over 333 MHz. (The Appendix describes an optimized shift register designed using MC' running internally at over 400 MHz in 2 μm CMOS.)

To achieve maximum throughput when an architecture contains loops (i.e., has "memory"), the MC-graph must be carefully designed to avoid bottlenecks in the data flow. To do this, it is sometimes necessary, if performance is critical, to add extra MC's into a path to act as buffers. This is analogous to placing extra pipeline latches to improve throughput in a synchronous system. In the three examples below, there are no bottlenecks, so they run at the maximum rate allowed by the technology, independent of the number of stages used.

It is suggested that on first reading, the reader should concentrate on the example which (s)he best understands (e.g., Section VII-A1, Section VII-A2, or Section VII-B) and just skim the other two.

A. Asynchronous Serial Parallel Multiplier

Multiplication is a common operation required for signal processing. There are many ways to implement this in hardware, but a popular [18]–[20] one is the Serial Parallel Multiplier (SPM). Fig. 11(a) shows a block diagram of a four-stage systolic SPM. It consists of a chain of four identical elements, one for each bit of the multiplicand. Each element, as shown in Fig. 11(b), has an input and an output for reading and writing one bit of the product (A), the multiplier (B), and the multiplicand (D) from its nearest neighbors. One reason this systolic version of the SPM is so attractive is that the clock is the only global signal.

The wavefront version of the systolic SPM is the same, with the added benefit of having no global signals of any kind. With four-state coding, each line in Fig. 11 is physically two lines going in the direction of the arrow (for parity and data) and an acknowledge (for phase) going in the opposite direction. Each element is composed of a number of MC's. The MC-graph depends on whether we want to do polynomial multiplication [see Fig. 13(a)] or integer multiplication [see Fig. 13(b)].

1) Polynomial Multiplier Element: This section describes the design of an Asynchronous Polynomial SPM element. A typical application of this type of multiplier is in error correction coding [21]. Polynomial multiplication is a multiply without carries. For example, the product (s) of two three-bit polynomials (the multiplicand d and the multiplier b) is

$$\begin{aligned}
s4 &= (d2 \text{ AND } b2)\\
s3 &= (d2 \text{ AND } b1) \text{ XOR } (d1 \text{ AND } b2)\\
s2 &= (d2 \text{ AND } b0) \text{ XOR } (d1 \text{ AND } b1) \text{ XOR } (d0 \text{ XOR } b2)\\
s1 &= (d1 \text{ AND } b0) \text{ XOR } (d0 \text{ AND } b1)\\
s0 &= (d0 \text{ AND } b0)
\end{aligned}$$

where

$$\begin{aligned}
s &= s4 \cdot x^4 + s3 \cdot x^3 + s2 \cdot x^2 + s1 \cdot x^1 + s0 \cdot x^0\\
d &= d2 \cdot x^2 + d1 \cdot x^1 + d0 \cdot x^0\\
b &= b2 \cdot x^2 + b1 \cdot x^1 + b0 \cdot x^0
\end{aligned}$$

and x is an indeterminate; "+" and "$\cdot$" refer to the addition and multiplication operators; and AND and XOR refer to the logical AND and exclusive-OR operators.

Fig. 13(a) shows one element of the asynchronous polynomial SPM built using just four MC's. The MC at the top of Fig. 13(a), labeled "P", operates the same as the shift register cell (C11) described earlier. After everything has been reset, the n-bit multiplicand (D) is loaded serially from the left, least significant bit (lsb) first. Every element of the SPM accepts a bit of the multiplicand on "di," passes this through its MC, then out onto "do" for the next element: passing the multiplicand

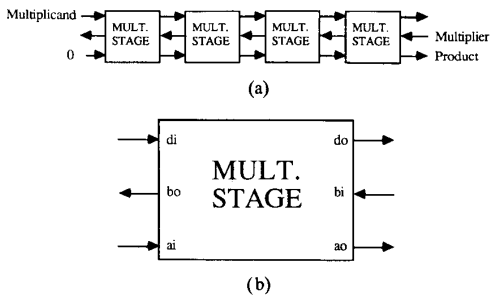


Fig. 11. Asynchronous serial parallel multiplier. (a) Four-bit serial parallel multiplier. (b) One serial parallel multiplier stage.

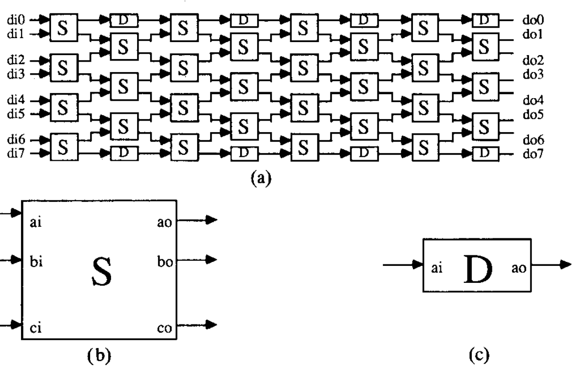


Fig. 12. Asynchronous bubble sorter. (a) 8-bit bubble sorter. (b) 2-bit sorter stage. (c) Delay/buffer stage.

down the line from left to right. This continues on down the chain, exactly as in the shift register loading process, until the lsb reaches the rightmost cell. Since the rightmost cell receives no acknowledge, it blocks the end of the chain. After n bits are loaded into the multiplicand (d) line, we know each element has one bit of the multiplicand stored in their respective MC's with the lsb stored in the rightmost element of the chain.

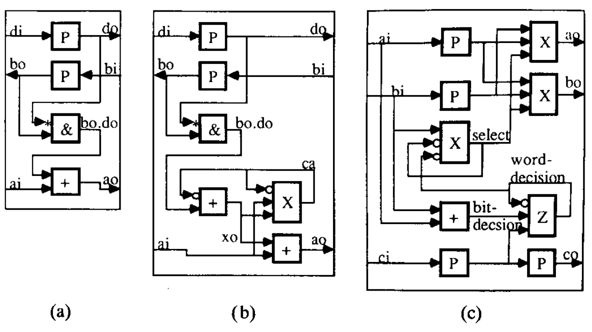
With the multiplicand in place, the multiplier is loaded in from the right, also lsb first. The rightmost cell receives this on its "bi" line, passes it through a second MC (also labeled P-MC), then out onto "bo" for the next element [see the second row from the top of Fig. 13(a)].

The rightmost element now ANDs together the lsbs of the multiplier and multiplicand and stores this result in a third MC (labeled "&"). Note that the multiplicand input is constant. Therefore, the &-MC must ignore its phase information (which is represented by the * in Fig. 13). The next bit of the multiplier is loaded after both the next cell's P-MC and its own &-MC have acknowledged they no longer need to see the lsb.

The final MC does an exclusive-OR (XOR) mapping to generate the partial product by calculating

$$ao = ai \text{ XOR } (bo \text{ AND } do).$$

By following through the flow of data, it can be seen that the output of the a-line of the SPM will be the polynomial product: coming out serially lsb first.



Notation	Multipliers	Sorter
ai	Partial Product Input	Data Input
ao	Partial Product Output	Most Significant Bit Output
bi	Multiplier Input	Data Input
bo	Multiplier Output	Least Significant Bit Output
ca	Carry Save	-
ci	-	Control Input
co	-	Control Output
di	Multiplicand Input	-
do	Multiplicand Output	-

o = Invert parity (ie pd -> p'd), * = ignored parity

Fig. 13. One stage of wavefront array built from MC's. (a) Polynomial multiplier. (b) Integer multiplier. (c) Packet sorter.

2) Integer Multiplier Element: This section describes the design of an asynchronous integer SPM element, performing part of a general multiply with carries between levels. Fig. 13(b) shows one element built with six MC's. The function of the top three MC's is identical to that for the polynomial multiplier, so we shall continue the description after the lsbs have been ANDed together.

The MC below the &-MC, labeled "+", performs an exclusive-OR (XOR) mapping whose output is the intermediate sum

$$xo = (bo \text{ AND } do) \text{ XOR } ca$$

where ca is the carry from the previous cycle. The circle at the ca input, which represents phase inversion, is necessary because the ca result of the previous cycle will have the opposite phase to the other input.

Having xo allows us to calculate the sum, ao, with an XOR MC. The output of this MC, which is shown at the bottom of Fig. 13(b), is

$$ao = ai \text{ XOR } (bo \text{ AND } do) \text{ XOR } ca$$
$$ao = ai \text{ XOR } xo.$$

The final MC, which uses a full multiplexer mapping (X-MC), calculates and stores the carry, ca, for the next cycle. Thus,

$$ca = (ai \text{ AND } (bo \text{ AND } do)) \text{ OR } (ca \text{ AND } (bo \text{ AND } do)) \text{ OR } (ca \text{ AND } ai)$$
$$ca = (ca \text{ AND } xo') \text{ OR } (ai \text{ AND } xo).$$

By inverting the parity of the ca feedback, we are able to make the X-MC act like a carry register.

By following through the flow of data, it can be seen that the output of the a-line of the SPM is the integer product: coming out serially lsb first.

B. An Asynchronous Sorter

Sorting numbers into ascending order is a common operation required for signal processing. There are many alternative techniques, trading speed, area, and complexity. We chose to implement a simple exchange sorter, similar to the bubble sorter [22]. To improve efficiency, more efficient designs, such as the Batcher sorter [23], [24], can be built using the same 2×2 sorting element.

Fig. 12(a) shows a block diagram of an eight-input sorter. It consists of four rows and eight columns of the basic 2×2 sort element, with each element having three inputs (ai, bi, and ci) outputs (ao, bo, co). The a and b lines hold two serial streams of data, while the c line delimits the packets. The c signal travels horizontally from cell to cell, so it is not drawn explicitly in Fig. 12(a). The wavefront version of this sorter has no global signals of any kind.

1) Sorter Element: Fig. 13(c) shows the MC-graph for the bit-level wavefront sort element, based on a synchronous packet sorter designed at Bellcore [25]. The two main serial inputs (ai and bi) arriving most significant bit (msb) first. The function of the cell is to route the input with the largest magnitude to the bottom (bo) and the smallest to the top (bo).

There are two "memory" cells in the packet sort element. The first, with select as its output, determines whether the data input (ai or bi) pass or cross. The second, with word-decision as its output, determines whether a decision has been made. The output select only changes state when word-decision is low.

Initially assume everything is reset, so that all outputs are low. The element is ready to accept the two data streams (A and B) and a control stream (C). The five MC's in the first column of Fig. 13(c) change state first. The top two MC's and the bottom MC (all labeled "P") buffer their respective inputs ("pipelining" to improve throughput). At the "same time" (though each cell keeps its own pace), the fourth cell (labeled "+") exclusive-ORs the two data inputs. Its output, bit-decision, is high only if the two input bits are different. The middle cell in the first column (labeled "X") implements the full multiplexer mapping. Since its control input, word-decision, is low, it passes the bi onto its output (select).

The second column of MC's now changes state. The bottom MC acts as another buffer for the c input. The top two MC's do the real data switching. Thus,

$$ao = bi \quad \text{and} \quad bo = ai, \qquad \text{if select was high;}$$
$$ao = ai \quad \text{and} \quad bo = bi, \qquad \text{if select was low.}$$

Since the control input, select, is equal to bi on the first cycle, the inputs will be swapped if bi was high. The final MC (labeled Z) performs the mapping

$$\text{word-decision} = ci' \text{ AND } (\text{word-decision OR bit-decision}).$$

Its output tells the next wave of data whether a decision has been made. A decision has been made if either a decision had previously been made (the old word-decision was high) or the two inputs this time were different (bit-decision is high), provided this is not the last bit in the packet (ci, is low).

When the second column has latched the first wave of data, the first column of MC's see their acknowledge inputs (not explicitly drawn on Fig. 13) change. The first row is then ready to repeat the process with a new wave of data; except that, if a decision was made the first time, the select output will remain fixed.

If we put two serial streams on the two data inputs (A and B), the biggest will come out on "ao" and the smallest on "bo." At any time we can put in a new packet, without flushing out or resetting, by setting the control input (ci) high at the end of the last packet. If ci were to remain high, making each packet one bit long, word-decision would remain permanently low. While, if ci were to remain low, after word-decision went high (after the first time the two data inputs were different), word-decision and select would remain fixed thereafter.

VIII. Practical Comparison

This section compares our four-state wavefront multipliers and sorter, with the equivalent clocked systolic arrays. The comparison is based on CMOS implementation, but in general is applicable to other technologies.

A. Design Complexity

The asynchronous approach we adopted, using just one type of cell (MC), leads to a much faster design and layout than for the equivalent synchronous approach. Even if we had predesigned synchronous standard cells, there are still more cell types to route together making the placement more difficult. Also, because there is only one cell, MC can be redesigned quicker than a standard cell library. Limiting ourselves to a multiplexer with a latch at its output in a synchronous design will approach, though not equal (because of the clock distribution), the simplicity of our MC-based asynchronous designs; however, the area of the synchronous array would grow significantly.

B. Speed

The difference in throughput between the equivalent optimized synchronous and asynchronous arrays depends on the architecture being implemented. From our experience with simulations and implementations of 2 μm CMOS implementations, such as those described in the last section, we found there was typically a 50% throughput (equivalent to clock speed) advantage in favor of the asynchronous array. In general, the more complex the basic cell, the greater the speed advantage of asynchronous arrays. Simulations (see Appendix) at higher levels of integration with a correspondingly larger number of cells showed that there was an increased speed advantage to the asynchronous approach.

C. Area

The area overhead of asynchronous arrays varied from around two times (for both multipliers) to almost six times (for the sorter). The reasons include using:

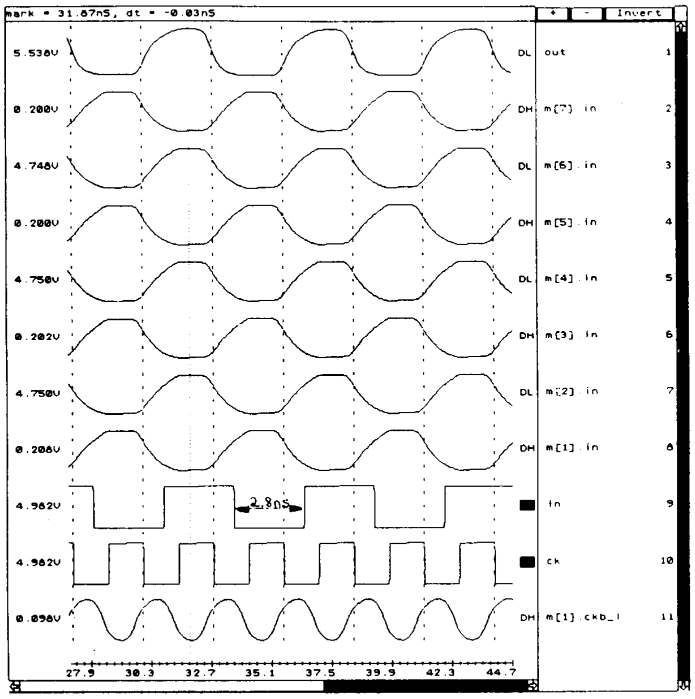


Fig. 14. Timing diagram for a synchronous shift register.

1) Three wires (two data and one acknowledge) for every bit of information,
2) Two latches for every bit of information stored,
3) Redundant logic and logic less amenable to minimization,
4) A latch on every gate (not necessary, but desirable for speed),
5) Only MC (not necessary, but desirable for reducing complexity).

Nearly half the area overhead associated with the bit-level asynchronous designs is caused by the handshaking overhead. Only a small part of the area difference is because of our decision to use only the MC. Not surprisingly, just as the relative speed improves as cells get more complex, so their relative area increases.

D. Reliability

Comparing robustness is not as straightforward as with the previous three parameters. More work is needed before we can come to any definite conclusions; particularly about the relative benefits of fault tolerance. It is likely, however, that for high-performance applications at high levels of integration, wavefront arrays will be significantly more reliable [26].

IX. Conclusion

This paper describes a modified approach to wavefront array design. Using a denser four-state code allows double the throughput, compared to the traditional three-state code. Using only multiplexing allows reduced complexity: to well below that of even a synchronous systolic array. Finally, by carefully optimizing the asynchronous multiplexer, the throughput is beyond that of even a carefully customized synchronous array.

The area penalty is a major drawback to our technique (and asynchronous designs in general); however, area is not always critical, particularly when a chip is pin limited. The area may be reduced by trading it for speed or complexity; however, without moving from bit-level to word-level structures, these techniques do significantly change the area overhead.

We believe that asynchronous logic, such as that described here, should be considered whenever speed, design complexity, or reliability are more important than area. Certainly, the almost universal use of synchronous logic in todays systems needs to be reexamined in the light of faster switching speeds and greater integration. Though this area/transistor overhead limits asynchronous logic at current levels of technology, we believe the benefits will become increasingly attractive for high-performance applications. Though the ideas have been principally described for on chip architectures, we believe the same motivation makes a modified form of this four-state asynchronous techniques applicable interchip communication (modified to remove the long acknowledge path and reduce the number of pins). Indeed, because of the longer, slower communication path, it may have far more applications for interchip communications.

Appendix
Shift Register Simulation

This Appendix describes the simulations and implemen-

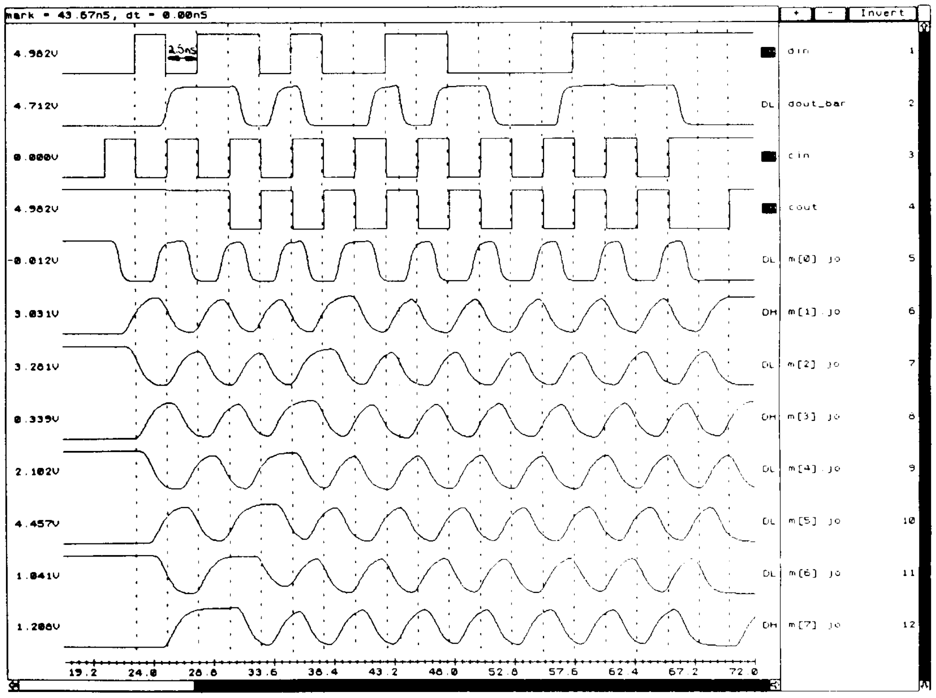


Fig. 15. Timing diagram for an asynchronous shift register.

tation of two shift registers to highlight some differences between asynchronous and synchronous arrays. Both the synchronous and asynchronous shift registers were designed using a dynamic logic implementation, with the layouts optimized for speed. The simulation results were obtained using a SPICE like simulator in 2 μm CMOS, and confirmed using a test-chip.

The dynamic synchronous shift register was implemented using nonminimum sized transistors (p devices were 16 μm wide and the n devices were 8 μm wide) and large clock drivers. The delay in the clock drivers was calculated and the input signal was delayed, so the throughput would not be too influenced by the delay in the buffers. The resulting simulated timing diagram for an eight-stage shift register is shown in Fig. 14. It shows the data as it passes from the input, through each stage of the register ($m[j]$.in, is the input to the jth stage of the register) and on to the output. The maximum frequency we could clock data was about 350 MHz.

Incorporating dynamic logic into an asynchronous system is difficult. With a clocked system, each storage node is refreshed every clock cycle; without a clock we must guarantee storage nodes are refreshed. Therefore, in addition to avoiding races and hazards, we must guarantee minimum state refresh. The implementation details are omitted here, but the dynamic asynchronous shift register was implemented using the same sized transistors as for the synchronous shift register. The timing diagram for an eight-state asynchronous shift register is shown in Fig. 15. It shows the input and output data, together with the acknowledges from each stage along the pipeline. The maximum frequency we could clock data was just over 400 MHz.

Comparing the two timing diagrams shows that the output changes in the synchronous shift register much more closely synchronized. Though the power consumption of the asynchronous shift register was almost twice that of the synchronous shift register, the peaks were greater for the synchronous case.

Clearly this one example does not prove asynchronous logic is faster. However, the shift register case is expected to be the worst for asynchronous logic, since the overhead of detecting when to change data is largest. Also, with more advanced technology simulations the differences were greater. Therefore, these results do confirm some of the potential for speed claimed in the paper.

Acknowledgment

I would like to thank C. J. Cotton for his considerable help and encouragement; also I wish to acknowledge the helpful suggestions of K. A. Kramer, R. M. F. Goodman, G. A. Hayward, J. N. Giacopelli, and the anonymous reviewers.

References

[1] C. L. Seitz, "System timing" in *Introduction to VLSI Systems,* Mead and Conway, Eds. Reading, MA: Addison-Wesley, 1980, ch. 7.
[2] H. T. Kung, "Why systolic architectures?" *IEEE Comput. Mag.,* vol. 15, pp. 97–107, Jan. 1982.
[3] ——, "Let's design algorithms for VLSI systems," Dep. Comput. Sci., Carnegie-Mellon Univ., Apr. 1983.
[4] J. Fortes and B. W. Wah, "Systolic arrays—From concept to implementation," *IEEE Comput. Mag.,* vol. 20, pp. 12–17, July 1987.
[5] C. L. Seitz, "Self-timed VLSI systems," in *Proc. Caltech Conf. VLSI,* Jan. 1979, pp. 345–353.

[6] I. Sutherland, "Micropipelines," *Commun. ACM,* vol. 32, pp. 720–738, June 1989.

[7] S. Y. Kung, R. J. Gal-Ezer, and K. S. Arun, "Wavefront array processors: Architecture, language and applications," in *Proc. Conf. Advanced Res. VLSI,* MIT, Jan. 1982, pp. 4–18.

[8] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront array processors—Concept to implementation," *IEEE Comput. Mag.,* vol. 20, pp. 18–33, July 1987.

[9] D. L. Dill and E. M. Clarke, "Automatic verification of asynchronous circuits using temporal logic," in *Proc. Chapter Hill Conf. VLSI,* 1985, pp. 127–140.

[10] A. J. Martin, "The design of a self-timed circuit for distributed mutual exclusion," in *Proc. Chapter Hill Conf. VLSI,* 1985, pp. 245–260.

[11] ——, "Compiling communicating processes into delay-insensitive VLSI circuits," *Distributed Comput.,* pp. 226–234, Jan. 1986.

[12] A. L. Fisher and H. T. Kung, "Synchronizing large VLSI arrays," *IEEE Trans. Comput.,* vol. C-34, pp. 734–740, Aug. 1985.

[13] D. M. Chapiro, "Reliable high speed arbitration and synchronization," *IEEE Trans. Comput.,* vol. C-36, pp. 1251–1255, Oct. 1987.

[14] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *IEEE J. Solid State Circuits,* vol. SC-24, pp. 62–70, Feb. 1989.

[15] J. A. Abraham *et al.,* "Fault tolerance techniques for systolic arrays," *IEEE Comput. Mag.,* vol. 20, pp. 65–74, July 1987.

[16] R. M. Goodman and A. J. McAuley, "An efficient asynchronous multiplier," in *Proc. Int. Conf. Systolic Arrays,* San Diego, CA, May 1988, pp. 593–600.

[17] R. Bannister and D. Whitehead, *Fundamentals of Digital Systems.* London, England: McGraw-Hill, 1973.

[18] I-N Chen and R. Willoner, "An O(n) parallel multiplier with bit sequential input and output" *IEEE Trans. Comput.,* vol. C-28, pp. 721–727, Oct. 1979.

[19] R. Gnanasekarean, "A fast serial-parallel binary multiplier" *IEEE Trans. Comput.,* vol. C-34, pp. 741–744, Aug. 1985.

[20] I-Chen Wu, "A fast 1-D serial parallel systolic multiplier," *IEEE Trans. Comput.,* vol. C-36, pp. 1243–1247, Oct. 1987.

[21] R. J. McEliece, *The Theory of Information and Coding.* Reading, MA: Addison-Wesley, 1977.

[22] D. E. Knuth, *The Art of Computer Programming: Vol. 3 Sorting and Searching,* 2nd ed. Reading, MA: Addison-Wesley, 1981.

[23] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comput. Conf.,* 1968, pp. 307–314.

[24] C. Day, J. Giacopelli, and J. Hickey, "Applications of self-routing switches to LATA fiber optic networks," in *Proc. Int. Switching Symp.,* Phoenix, AZ, 1987, pp. A7.3.1–A7.3.5.

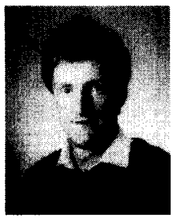
[25] W. S. Marcus, "A CMOS Batcher and Banyan chip set for B-ISDN packet switching," *IEEE J. Solid State Circuits,* vol. SC-25, Dec. 1990.

[26] R. M. Goodman, K. Kramer, and A. J. McAuley, "The inherent fault tolerance of asynchronous arrays," in *Proc. Int. Conf. Systolic Arrays,* Killarney, May–June 1989, pp. 567–576.

[27] M. Hatamian and G. L. Cash, "A 70-MHz 8-bit $\times$ 8-bit parallel pipelines multiplier in 2.5 μ CMOS," *IEEE J. Solid State Circuits,* vol. SC-21, pp. 503–513, Aug. 1986.

[28] K. K. Parhi and M. Hatamian, "A high sample rate recursive digital filter chip," *VLSI Signal Processing III,* IEEE, Nov. 1989.

[29] G. Jacobs and R. Brodersen, "A fully-asynchronous digital signal processor using self-timed circuits," in *Proc. ISSCC Dig. Tech. Papers,* Feb. 1990, pp. 150–151.



Anthony J. McAuley was born in Liverpool, England. He received the B.S. and Ph.D. from Hull University, England, in 1981 and 1985, respectively.

From 1985 to 1987 he was a Postdoctoral Research Fellow at Caltech, Pasadena, CA. Since 1987 he has worked with the Computer Communications Research District in Bellcore, Morristown, NJ. His active research interests include packet communication, error control, self-timed systems, VLSI architecture, and computer arithmetic.