INVITED PAPER

Asynchronous Techniques for System-on-Chip Design

Digital circuit designs that are not sensitive to delay promise to allow operation without clocks for future systems-on-a-chip.

By Alain J. Martin, *Member IEEE*, and Mika Nyström, *Member IEEE*

ABSTRACT | SoC design will require asynchronous techniques as the large parameter variations across the chip will make it impossible to control delays in clock networks and other global signals efficiently. Initially, SoCs will be globally asynchronous and locally synchronous (GALS). But the complexity of the numerous asynchronous/synchronous interfaces required in a GALS will eventually lead to entirely asynchronous solutions. This paper introduces the main design principles, methods, and building blocks for asynchronous VLSI systems, with an emphasis on communication and synchronization. Asynchronous circuits with the only delay assumption of isochronic forks are called quasi-delay-insensitive (QDI). QDI is used in the paper as the basis for asynchronous logic. The paper discusses asynchronous handshake protocols for communication and the notion of validity/neutrality tests, and completion tree. Basic building blocks for sequencing, storage, function evaluation, and buses are described, and two alternative methods for the implementation of an arbitrary computation are explained. Issues of arbitration, and synchronization play an important role in complex distributed systems and especially in GALS. The two main asynchronous/synchronous interfaces needed in GALS—one based on synchronizer, the other on stoppable clock—are described and analyzed.

KEYWORDS | Arbiter; asynchronous; asynchronous bus; asynchronous/synchronous interface; C-element; completion tree; dual-rail; globally asynchronous and locally synchronous (GALS); half-buffer; handshake protocol; isochronic fork; metastability; passive–active buffer; precharge half-buffer (PCHB); quasi-delay-insensitive (QDI); stoppable clock; synchronizer

Manuscript received September 14, 2005; revised March 3, 2006. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA).
The authors are with the Department of Computer Science, California Institute of Technology, Pasadena, CA 91125 USA (e-mail: alain@async.caltech.edu).

Digital Object Identifier: 10.1109/JPROC.2006.875789

I. INTRODUCTION

It is now generally agreed that the sizable very large scale integration (VLSI) systems [*systems-on-chip* (SoCs)] of the nanoscale era will not operate under the control of a single clock and will require asynchronous techniques. The large parameter variations across a chip will make it prohibitively expensive to control delays in clocks and other global signals. Also, issues of modularity and energy consumption plead in favor of asynchronous solutions at the system level. Whether those future systems will be entirely asynchronous, as we predict, or globally asynchronous and locally synchronous (GALS), as more conservative practitioners would have it, we anticipate that the use of asynchronous methods will be extensive and limited only by the traditional designers' relative lack of familiarity with the approach. Fortunately, the past two decades have witnessed spectacular progress in developing methods and prototypes for asynchronous (clockless) VLSI. Today, a complete catalogue of mature techniques and standard components, as well as some computer-aided design (CAD) tools, are available for the design of complex asynchronous digital systems.

This paper introduces the main design principles, methods, and building blocks for asynchronous VLSI systems, with an emphasis on communication and synchronization. Such systems will be organized as distributed systems on a chip consisting of a large collection of components communicating by message exchange. Therefore, the paper places a strong emphasis on issues related to network and communication—issues for which asynchronous techniques are particularly well-suited. Our hope is that after reading this paper, the designer of an SoC should be familiar enough with those techniques that he or she would no longer hesitate to use them. Even those adepts of GALS who are adamant not to let asynchrony penetrate further than the network part of their SoC must realize that network architectures for SoCs are rapidly becoming so complex as to require the mobilization of the complete armory of asynchronous techniques.

The paper is organized as follows. The second section contains a brief history and the main definitions of the different asynchronous logics according to their timing assumptions. Section III introduces the computational models and languages used in this paper to describe and construct asynchronous circuits. Section IV introduces the most common asynchronous communication protocols and the notion of validity/neutrality tests. Basic building blocks for sequencing, storage, and function evaluation are introduced in Section V. Section VI presents two alternative methods for the implementation of an arbitrary computation: syntax-directed decomposition and data-driven decomposition. The two approaches differ in how a specification is decomposed into pipeline stages. Section VII describes several implementations of buses. Section VIII deals with issues of arbitration, and synchronization. Section IX presents the asynchronous/synchronous interfaces needed in a GALS system.

II. A BRIEF HISTORY AND A FEW DEFINITIONS

The field of asynchronous design is both old and new. The 1952 ILLIAC and the 1962 ILLIAC II at the University of Illinois are said to have contained both synchronous and asynchronous parts [2]. The 1960 PDP6 from Digital Equipment (DEC) was also asynchronous [52]. The "macromodule" experiment in the 1960s proposed asynchronous building blocks that, in spirit, are very close to a modern system-level approach [10] and to GALS. Important theoretical contributions of this period include the works of Huffman [20], Muller [33], and Unger [53]. An excellent presentation of Muller's work is in [32]. The pioneering work of Molnar and his colleagues at Washington University was instrumental in explaining metastability [6].

Even though asynchronous logic never disappeared completely, when clocked techniques offered an easy way of dealing with timing and hiding hazards, clockless logic was all but forgotten until the arrival of VLSI in the late 1970s. The first Caltech Conference on VLSI in 1979 contained a complete session on "self-timed" logic, as asynchronous logic was called at the time, with in particular an important paper by Stucki and Cox on "synchronization strategies" presenting the first *pausable clock*—as we shall see, an important device in GALS—and discussing metastability [47]. Seitz's chapter on system timing in Mead and Conway's epoch-making 1980 *Introduction to VLSI Systems* revived the research community's interest in the topic—if not the industry's interest [31], [45].

The first "modern" synthesis methods appear around the mid1980 with the Caltech program-transformation approach [24] and T.-A. Chu's State-transition-graph (STG) approach [8]. Soon after, Burns and Martin, Brunvand, and van Berkel proposed similar methods for the syntax-directed compilation of high-level description into asynchronous circuits [3], [5]. *Petrify* is a more recent tool for the synthesis of asynchronous controllers described as Petri nets [12].

The first single-chip asynchronous microprocessor was designed at Caltech in 1988 [27]. It was followed by the first "Amulet" (a family of asynchronous clones of the ARM processor) from the University of Manchester in 1993 [14], the TITAC, an 8-bit microprocessor from the Tokyo Institute of Technology in 1994 [36], and the Amulet2e and TITAC-2 in 1997 [15], [37]—the TITAC-2 is a 32-bit microprocessor. Also in 1997, the Caltech group designed the MiniMIPS, an asynchronous version of the 32-bit MIPS R3000 microprocessor [28]. With a performance close to four times that of a clocked version in the same technology for the first prototype, the MiniMIPS is, at the moment of writing, still the fastest complete asynchronous processor ever fabricated [30]. A group at Grenoble ported the Caltech MiniMIPS building blocks to standard cells to use in a 16-bit RISC [42]. Other asynchronous chip experiments include the design of a fast divider at Stanford in 1991 [49], and an instruction-length decoder for the Pentium by a research group at Intel in 1999 [44]. Low-power asynchronous microcontrollers have been designed at Philips [17], Caltech [30], and Cornell [21]. The concept of GALS was first proposed by Chapiro [7] in 1984. It has recently gained in popularity, in particular with the work of a group at Zurich [35].

A digital circuit is asynchronous when no clock is used to implement sequencing. Such circuits are also called clockless.

Several books on asynchronous logic have been published. Among them, our favorites are [11], [34], and [46]. A special issue of the *Proceedings of the IEEE* also gives a good overview of the state of the art [38]. The book by Dally and Poulton, although not specifically about asynchronous systems contains an excellent chapter on synchronization [13]. The on-line *Asynchronous Bibliography* maintains an updated bibliography of the field to date [1].

A digital circuit is *asynchronous* when no clock is used to implement sequencing. Such circuits are also called *clockless*. The various asynchronous approaches differ in their use of delay assumptions to implement sequencing. *A circuit is delay-insensitive (DI) when its correct operation is independent of any assumption on delays in operators and wires except that the delays are finite and positive*. The term delay-insensitive appears informally in [45]. It was proved in 1990 that in a model in which all delays are exposed—the building blocks are elementary gates with a single

Boolean output—the class of entirely delay-insensitive circuits is very limited [26]. Most circuits of interest to the digital designer fall outside the class. But it can also be proved that a single delay assumption on certain forks connecting the output of a gate to the inputs of several other gates is enough to implement a Turing machine, and therefore the whole class of Turing-computable functions [23]. Those forks are called *isochronic*.

Asynchronous circuits with the only delay assumption of isochronic forks are called quasi-delay-insensitive (QDI). We use QDI as the basis for asynchronous logic. All other forms of the technology can be viewed as a transformation from a QDI approach by adding some delay assumption. An asynchronous circuit in which *all* forks are assumed isochronic corresponds to what has been called a *speed-independent* circuit, which is a circuit in which the delays in the interconnects (wires and forks) are negligible compared to the delays in the gates. The concept of speed-independent circuit was introduced by Muller [33]. Similarly, *self-timed* circuits are asynchronous circuits in which all forks that fit inside a chosen physical area called *equipotential region* are isochronic [45].

Several styles of asynchronous circuits currently in use fall into some hybrid category. They rely on some specific timing assumption besides the implicit QDI assumption. For instance, the "bundled data" technique uses a timing assumption to implement the communication protocol between components. Another approach—*timed asynchronous logic*—starts from a QDI circuit, and then derives timing relations between events in the QDI computation that are used to simplify the solution. (See [34].) Yet another approach uses a timing assumption to control the reset phase of the handshake protocol (to be explained later). Two logic families based on this approach are *asynchronous pulse logic* ([40]) and GasP ([48]).

III. SoCs AS DISTRIBUTED SYSTEMS

SoCs are complex distributed systems in which a large number of parallel components communicate with one another and synchronize their activities by message exchange. At the heart of digital logic synthesis lie fundamental concurrency issues like concurrent read and write of variables and synchronization between send and receive commands. Synchronous (clocked) logic brings a simple solution to the problem by partially ordering transitions with respect to a succession of global events (clock signals) so as to order conflicting read/write actions. In the absence of a global time reference, asynchronous logic has to deal with concurrency in all its generality, and asynchronous-logic synthesis relies on the methods and notations of concurrent computing. There exist many languages for distributed computing. The high-level language used in this paper is called *Communicating Hardware Processes* (CHP). It is based on CSP [19], and is used widely in one form or other in the design of asynchronous systems. We introduce only those constructs of the language needed for describing the method and the examples, and that are common to most computational models based on communication. The systematic design of an SoC is a process of successive refinements taking the design from a high-level description to a transistor netlist. The three levels of representation—CHP, HSE, PRS—used in this paper mirror the three main stages of the refinement.

A. Modeling Systems: Communicating Processes

A system is composed of concurrent modules called *processes*. Processes do not share variables but communicate only by send and receive actions on ports.

1) Communication, Ports, and Channels: A send port of a process—say, port R of process $p1$—is connected to a receive port of another process—say, port L of process $p2$—to form a *channel*. A receive command on port L is denoted $L?y$. It assigns to local variable y the value received on L. A send command on port R, denoted $R!x$, assigns to port R the value of local variable x. The data item transferred during a communication is called a message. The net effect of the combined send $R!x$ and receive $L?y$ is the assignment $y := x$ together with the synchronization of the send and receive actions.

The *slack* of a channel is the maximal difference between the number of completed send actions and the number of completed receive actions on the two ports of the channel. In other words, the slack is the capacity of the channel to store messages. Since we implement channels with wires only, we choose to have *slack-zero channels*: the completion of a send at one end of the channel coincides with the completion of a receive at the other end of the channel. Both send and receive actions are said to be "blocking." A send or receive action on a port may have to be delayed (pending) until the matching action on the other port of the channel is ready.

2) Assignment: The value of a variable is changed by an explicit *assignment* to the variable as in $x := expr$. For b Boolean, $b\uparrow$ and $b\downarrow$ stand for $b :=$ true and $b :=$ false, respectively.

3) Sequential and Parallel Compositions: CHP and HSE provide two composition operators: the sequential operator $S1; S2$ and the parallel operator. Unrestricted use of parallel composition would cause read/write conflicts on shared variables. CHP restricts the use of concurrency in two ways. The parallel bar $\|$, as in $S1\|S2$, denotes the parallel composition of processes.[1] CHP also allows a limited form of concurrency *inside* a process, denoted by the comma, as in $S1$, $S2$. The comma is restricted to

[1]In CHP, processes do not share variables. In HSE, the only shared variables are those introduced for the implementation of communication. They cannot cause read/write conflicts.

program parts that are *noninterfering*: if $S1$ writes x, then $S2$ neither reads x nor writes x.

4) Selection, Wait, and Repetition: The *selection command* $[B1 \rightarrow S1[]B2 \rightarrow S2[] \ldots]$ is a generalization of the if-statement. It has an arbitrary number (at least one) of clauses, called "guarded commands," $Bi \rightarrow Si$ where Bi is a Boolean condition and Si is a program part. The execution of the selection consists of: 1) evaluating all guards and 2) executing the command Si with the true guard Bi. In this version of the selection, at most one guard can be true at any time. There is also an arbitrated version where several guards can be true. In that case, an arbitrary true guard is selected. The arbitrated selection is identified by a thin bar as in $[B1 \rightarrow S1|B2 \rightarrow S2]$.

In both versions, when no guard is true, the execution is suspended: the execution of the selection reduces to a *wait* for a guard to become true. Hence, waiting for a condition to be true can be implemented with the selection $[B \rightarrow skip]$, where *skip* is the command that does nothing but terminates. A shorthand notation for this selection is $[B]$.

In this paper, we use only the nonterminating repetition $*[S]$ that repeats S forever.

5) Pipeline Slack and Slack Matching: Slack matching is an optimization by which simple buffers are added to a system of distributed processes to increase the throughput. A pipeline is a connected subgraph of the process graph with one input port and one output port. The *static slack* of a pipeline is the maximal number of messages the pipeline can hold. A pipeline consisting of chain of n simple buffers has a static slack of n, since each simple buffer can hold at most one message, and the channels have slack zero—unless, as we shall see, the buffers implementations are subjected to a transformation called reshuffling, which can reduce their slack.

The *dynamic slack* of a pipeline denotes the number of messages or, more generally, the range of numbers of messages that the pipeline must hold to run at optimal throughput. For the same pipeline of n simple buffers with a symmetric implementation, the dynamic slack is centered around $n/2$. (See [22], [41].) However, the most efficient buffer templates are not symmetrical—they favor forward latency over backward latency. For such buffers, the dynamic-slack range is reduced, typically centering around $n/8$ for the MiniMIPS. (The definitions of static and dynamic slacks are easy to extend to rings of processes.)

B. Modeling System Components: HSE

Each CHP process is refined into a partial order of signal transitions, i.e., transitions on Boolean variables. The HSE notation is not different from CHP except that it allows only Boolean variables, and send and receive communications have been replaced with their *handshaking expansion* in terms of the Boolean variables modeling the communication wires. The modeling of wires introduces a restricted form of shared variables between processes (the variables implementing channels). A typical example of an HSE program is

$$*[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ lo\downarrow].$$

The input variables li and ri can only be read. The output variables lo and ro can be read and written. The above example can be read as follows. "Repeat forever: wait for li to be true; set ro to true; wait for ri to be true; set ro to false; etc ..."

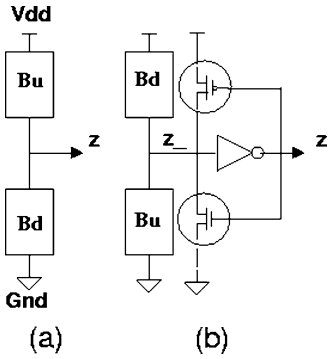

Fig. 1. *The pull-up and pull-down networks implementing a CMOS logic gate. (a) Combinational gate. (b) State-holding gate with a standard "staticizer" ("keeper"). The circled transistors are weak.*

C. Modeling Circuits: Production Rules

A circuit—for instance, a CMOS circuit—is a network of operators (logic gates). Each gate has an arbitrary number of inputs (in practice this number is limited by the resistance of transistor chains), and one output. (The only exceptions are arbiter and synchronizer, which each have two inputs and two outputs.) A logic gate with Boolean output z sets z to true when a Boolean condition Bu of its inputs holds, and sets z to false when a Boolean condition Bd of its inputs holds. Those behaviors are formalized by the two *production rules*

$$Bu \rightarrow z\uparrow$$
$$Bd \rightarrow z\downarrow.$$

A production rule (PR) is a construct of the form $B \rightarrow t$ where t is a simple assignment (a transition) and B is a Boolean expression called the guard of the PR. A production rule set (PRS) is the parallel composition of all production rules in the set. Each pair of complementary PRs,—the production rules that set and reset the same variable—is identified and implemented with (standard or nonstandard) CMOS operators. In order to carry over the robustness and delay insensitivity of the design to the circuit level, all operators are restoring (they have gain) and static (the output node is never "floating").

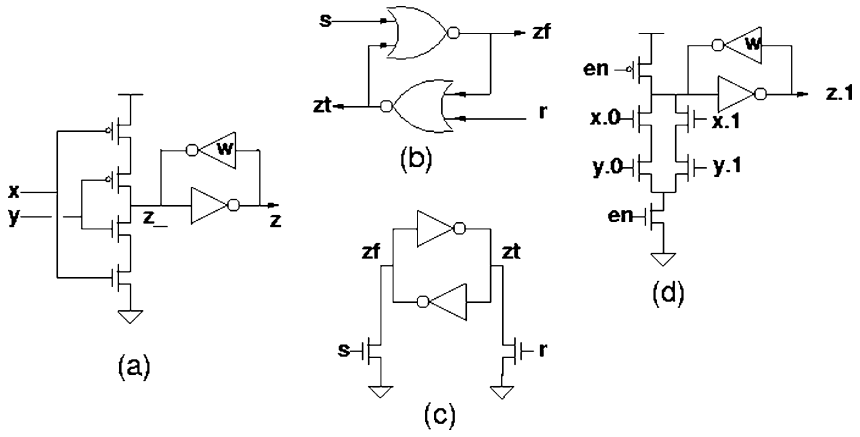

Fig. 2. *State-holding elements. (a) The C-element. (b) NOR-gate implementation of set–reset. (c) Inverter implementation of set–reset. (d) Example of a precharge function. The cross-coupled inverters in (a) and (b) are standards "staticizers." The letter w indicates that the transistors of the inverter are weak.*

Complementary PRs must be *noninterfering*, i.e., guards Bu and Bd cannot be true at the same time. If $Bu = \neg Bd$, then either Bu or Bd holds at any time, and output z is always connected to either the Vdd or the ground—z is always "driven." In this case, the operator is *combinational* with the simple CMOS implementation of Fig. 1(a). If there are states in the computation where neither Bu nor Bd holds, then output z is "floating" in those states. The operator has to maintain the current value of z and is therefore *state-holding*. In a QDI circuit, a state-holding operator is usually implemented with the "keeper" or "staticizer" shown in Fig. 1(b). (Adding a staticizer introduces interferences between the original PRs and the rules added by the feedback inverter. The interferences are resolved by making the transistors of the feedback inverter "weak," i.e., with low drain-to-source current.) Alternatively, the state-holding operator can be transformed into a combinational one, like the NOR-gate implementation of the set–reset shown in Fig. 2(b).

1) Combinational Gates: With the exception of the "write-acknowledge," we use standard combinational gates in this paper: inverter, NAND, NOR. NAND- and NOR-gates may have multiple inputs. Because of the restriction on the length of transistor chains, a multiple-input gate may have to be decomposed into a tree of smaller gates.[2]

2) State-Holding Elements: The three state-holding gates used in this paper are the Muller C-element, the set–reset gate, and the precharge function. The *Muller C-element* (also called C-element) with inputs x and y and output z, denoted $z = x\underline{C}y$, implements the PRs $x \wedge y \rightarrow z\uparrow$ and $\neg x \wedge \neg y \rightarrow z\downarrow$. The C-element is a state-holding element, since the current value of z must be maintained when $x \neq y$. The three-input C-element (denoted 3C) is used in a few examples in the paper. The *set–reset gate* with inputs s and r and output z has the PRs $s \rightarrow z\uparrow$ and $r \rightarrow z\downarrow$.

Since s and r must be mutually exclusive, it is always possible to implement the set–reset gate as the C-element $z = s\underline{C}\neg r$. This implementation requires to invert either s or r. Rather, we can code z with the two variables zt and zf, such that $zt = \neg zf$, and we implement the set–reset gate either with two cross-coupled NOR-gates or two cross-coupled inverters as shown in Fig. 2(b) and (c).

We also use a nonstandard operator, the *precharge function* (PCF). An example of a precharge function with inputs en, $x.0$, $x.1$, $y.0$, $y.1$ and output z is described by the following two PR pairs:

$$\begin{aligned} en \wedge (x.0 \wedge y.0 \vee x.1 \wedge y.1) &\rightarrow z.1\_\downarrow \\ \neg en &\rightarrow z.1\_\uparrow \\ en \wedge (x.0 \wedge y.1 \vee x.1 \wedge y.0) &\rightarrow z.0\_\downarrow \\ \neg en &\rightarrow z.0\_\uparrow. \end{aligned}$$

This gate computes the function $X = Y$ where X, Y, and Z are dual-rail encoded; en is a control signal. This gate is also a state-holding element. The PCF can be used for any function (Boolean or 1-of-N) whose CMOS pulldown-networks do not exceed the limit on transistor-chain length.

D. Stability and Noninterference

Stability and noninterference are the two properties of PRS that guarantee that the circuits are operating correctly, i.e., without *logic hazards*. A hazard is the possibility of an incomplete transition (a "glitch").[3]

[2]Large-gate decomposition is an annoying problem in asynchronous design, since it may violate stability. The decomposition of a multi-input OR-gate we use in the paper is stable because a transition on the output is caused by exactly one input transition.

[3]Of course, electrical effects, in particular charge sharing and cross-talk, can also produce voltage glitches that are not eliminated by stability and noninterference.

How do we guarantee the proper execution of production rule $G \rightarrow t$? In other words, what can go wrong and how do we avoid it? Two types of malfunction may take place: 1) G may cease to hold before transition t has completed, as the result of a concurrent transition invalidating G, and 2) the *complementary transition* t' of t is executed while the execution of t is in progress, leading to an undefined state. We introduce two requirements, *stability* and *noninterference* that eliminate the two sources of malfunction.

The "result" $R(t)$ of a transition t is defined as $R(x\uparrow) = x$, and $R(x\downarrow) = \neg x$, for any x.

Definition 1: A production rule $G \rightarrow t$ is said to be stable in a computation if and only if G can change from true to false only in those states of the computation in which $R(t)$ holds. A production-rule set is said to be stable if and only if all production rules in the set are stable.

Definition 2: Two production rules $Bu \rightarrow x\uparrow$ and $Bd \rightarrow x\downarrow$ are said to be noninterfering in a computation if and only if $\neg Bu \vee \neg Bd$ is an invariant of the computation. A production-rule set is noninterfering if every pair of complementary production rules in the set is noninterfering.

Any concurrent execution of a stable and noninterfering PRS is equivalent to the sequential execution model in which, at each step of the computation, a PR with a true guard is selected and executed. The selection of the PR should be weakly fair, i.e., any enabled PR is eventually selected for execution.

The existence of a sequential execution model for QDI computations greatly simplifies reasoning about, and simulating, those computations. Properties similar to stability are used in other theories of asynchronous computations, in particular, *semimodularity* [33] and *persistency* [11]. Consider rule $B \wedge x \rightarrow x\downarrow$ as part of a gate G with output x. At the logical level, the execution of transition $x\downarrow$ when the guard holds invalidates the guard. (Such production rules are therefore called *self-invalidating*.) *We exclude self-invalidating production rules, since, in most implementations, they would violate the stability condition.*

E. Isochronic Forks

A computation implements a partial order of transitions. In the absence of timing assumptions, this partial order is based on a causality relation. For example, transition $x\uparrow$ causes transition $y\downarrow$ in state S if and only if $x\uparrow$ makes guard By of $y\downarrow$ true in S. Transition $y\downarrow$ is said to *acknowledge* transition $x\uparrow$. We do not have to be more specific about the precise ordering in time of transitions $x\uparrow$ and $y\downarrow$. The acknowledgment relation is enough to introduce the desired partial order among transitions, and to conclude that $x\uparrow$ precedes $y\downarrow$. In an implementation of the circuit, gate Gx with output x is directly connected to gate Gy with output y, i.e., x is an input of Gy.

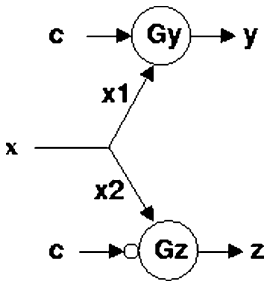

Fig. 3. *The fork (x, x1, x2) is isochronic: a transition on x1 causes a transition on y only when c is true, and a transition on x2 causes a transition on z only when c is false. Hence, certain transitions on x1 and on x2 are not acknowledged, and therefore a timing assumption must be used to guarantee the proper completion of those unacknowledged transitions.*

Hence, a necessary condition for an asynchronous circuit to be delay-insensitive is that all transitions are acknowledged.

Unfortunately, the class of computations in which all transitions are acknowledged is very limited. Consider the example of Fig. 3. Signal x is forked to $x1$, an input of gate Gy with output y, and to $x2$, an input of gate Gz with output z. A transition $x\uparrow$ when c holds is followed by a transition $y\uparrow$, but not by a transition $z\uparrow$, i.e., transition $x1\uparrow$ is acknowledged but transition $x2\uparrow$ is not, and *vice versa* when $\neg c$ holds. Hence, in either case, a transition on one output of the fork is not acknowledged. In order to guarantee that the unacknowledged transition completes without violating the specified order, a timing assumption called the *isochronicity assumption* has to be introduced, and the forks that require that assumption are called *isochronic forks* [26]. (Not all forks in a QDI circuit are isochronic.) Most circuits presented in this paper contain isochronic forks. A typical instance is the fork with input qi in Fig. 13 describing a single-bit register.

The timing assumption on isochronic forks is a one-sided inequality that can always be satisfied: it requires that the delay of a single transition be shorter that the sum of the delays on a multitransition path. It can be proved by constructing a Turing machine as a QDI circuit (see [23]) that the class of QDI circuits is Turing-complete, i.e., all Turing-computable functions have a QDI implementation.

IV. ASYNCHRONOUS COMMUNICATION PROTOCOLS

The implementation of send/receive communication is central to the methods of asynchronous logic, since this form of communication is used at all levels of system design, from communication between, say, a processor and a cache down to the interaction between the control part and the datapath of an ALU. Communication across a channel connecting two asynchronous components $p1$ and

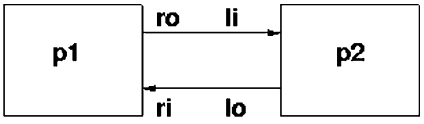

Fig. 4. *Implementation of a "bare" channel (L, R) with two handshake wires: (lo, ri) and (ro, li).*

$p2$ is implemented as a *handshake protocol*. In a later section, we will describe how to implement communication between a synchronous (clocked) component and an asynchronous one. Such interfaces are needed in a GALS SoC.

A. Bare Handshake Protocol

Let us first implement a "bare" communication between processes $p1$ and $p2$: no data is transmitted. (Bare communications are used as a synchronization point between two processes.) In that case, channel (R, L) can be implemented with two wires: wire (ro, li) and wire (lo, ri). (The wires that implement a channel are also called *rails*.) See Fig. 4.

Wire (ro, li) is written by $p1$ and read by $p2$. Wire (lo, ri) is written by $p2$ and read by $p1$. An assignment $ro\uparrow$ or $ro\downarrow$ in $p1$ is eventually followed by the corresponding assignment $li\uparrow$ or $li\downarrow$ in $p2$ due to the behavior of wire (ro, li). And symmetrically for variables lo and ri, and wire (lo, ri). By convention, and unless specified otherwise, all variables are initialized to **false**.

1) Two-Phase Handshake: The simplest handshake protocol implementing the slack-zero communication between R and L is the so-called *two-phase handshake* protocol, also called *nonreturn to zero* (NRZ). The protocol is defined by the following handshake sequence Ru for R and Lu for L:

$$\begin{aligned} Ru: &\quad ro\uparrow;\ [ri] \\ Lu: &\quad [li];\ lo\uparrow. \end{aligned}$$

Given the behavior of the two wires (ro, li) and (lo, ri), the only possible interleaving of the elementary transitions of Ru and Lu is $ro\uparrow; li\uparrow; lo\uparrow; ri\uparrow$.

This interleaving is a valid implementation of a slack-zero execution of R and L, since there is no state in the system where one handshake has terminated and the other has not started. But now all handshake variables are true, and therefore the next handshake protocol for R and L has to be

$$\begin{aligned} Rd: &\quad ro\downarrow;\ [\neg ri] \\ Ld: &\quad [\neg li];\ lo\downarrow. \end{aligned}$$

The use of the two different protocols is possible if it can be statically determined (i.e., by inspection of the CHP code) which are the even (up-going) and odd (down-going) phases of the communication sequence on each channel. But if, for instance, the CHP program contains a selection command, it may be impossible to determine whether a given communication is an even or odd one. In that case, a general protocol has to be used that is valid for both phases, as follows:

$$\begin{aligned} R: &\quad ro := \neg ro;\ [ro = ri] \\ L: &\quad [lo \neq li];\ lo := \neg lo. \end{aligned}$$

This protocol has a complicated circuit implementation, requiring exclusive-or gates and the storage of the current values of lo and ro. Two-phase handshake also requires that arithmetic and logical operations performed on the data transmitted be implemented in both upgoing and downgoing logics, which is quite inefficient. Therefore, in spite of its simplicity, the two-phase handshake protocol is rarely used besides some obvious cases.

2) Four-Phase Handshake: A straightforward solution is to always reset all variables to their initial value (zero). Such a protocol is called *four-phase* or *return-to-zero* (RZ). R is implemented as $Ru; Rd$ and L as $Lu; Ld$ as follows:

$$\begin{aligned} R: &\quad ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri] \\ L: &\quad [li];\ lo\uparrow;\ [\neg li];\ lo\downarrow. \end{aligned}$$

In this case, the only possible interleaving of transitions for a concurrent execution of R and L is $ro\uparrow; li\uparrow; lo\uparrow; ri\uparrow; ro\downarrow; li\downarrow; lo\downarrow; ri\downarrow$.

Again, it can be shown that this interleaving implements a slack-zero communication between R and L. It can even be argued that this implementation is in fact the sequencing of two slack-zero communications: the first one between Ru and Lu, the second one between Rd and Ld. This observation will be used later to optimize the protocols by a transformation called *reshuffling*.

B. Handshake Protocols With Data: Bundled Data

Let us now deal with the case when the communication also entails transmitting data, for instance, by sending on $R(R!x)$ and receiving on $L(L?y)$. A solution immediately comes to mind: let us add a collection of data wires next to the handshake wires. The data wire (rd, ld) is indicated by a double arrow on Fig. 5. The protocols are as follows:

$$\begin{aligned} R!x: &\quad rd := x;\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri] \\ L?y: &\quad [li];\ y := ld;\ lo\uparrow;\ [\neg li];\ lo\downarrow. \end{aligned}$$

This protocol relies on the timing assumption that the order between $rd := x$ and $ro\uparrow$ in the sender is maintained in the

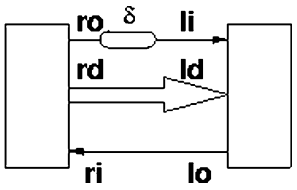

Fig. 5. *A bundled-data communication protocol. The cigar shape on the control wire (ro, li) indicates that the delay δ on the wire has been adjusted to be longer than the delays on the data wires (rd, ld).*

receiver: when the receiver has observed *li* to be true, it can assume that *ld* has been set to the right value, which amounts to assuming that the delay on wire (ro, li) is always "safely" longer than the delay on wire (rd, ld). Such a protocol is used and is called *bundled-data*. The efficiency of bundle-data versus DI codes is a hotly debated issue. We will discuss it later.

C. DI Data Codes

In the absence of timing assumptions, the protocol cannot rely on a single wire to indicate when the data wires have been assigned a valid value by the sender. The validity of the data has to be encoded with the data itself. A DI data code is one in which the validity and neutrality of the data are encoded within the data. Furthermore, the code is chosen such that when the data changes from neutral to valid, no intermediate value is valid; when the data changes from valid to neutral, no intermediate value is neutral. Such codes are also called *separable*. There are many DI codes but two are almost exclusively used on chip—the *dual-rail* and *1-of-N* codes.

D. Dual-Rail Code

In a dual-rail code, two wires, bit.0 and bit.1, are used for each bit of the binary representation of the data. [See Fig. 6(a).] The neutral and valid values are encoded as follows:

value :	neutral	0	1
bit.0 :	0	1	0
bit.1 :	0	0	1

For a two-bit data word $(x0, x1)$, its dual-rail encoding is

value :	neutral	0	1	2	3
$x0.0$:	0	1	0	1	0
$x0.1$:	0	0	1	0	1
$x1.0$:	0	1	1	0	0
$x1.1$:	0	0	0	1	1

E. 1-of-N Codes

In a *1-of-N code*, one wire is used for each value of the data. Hence, the same two-bit data word is now encoded as follows:

value :	neutral	0	1	2	3
$d.0$:	0	1	0	0	0
$d.1$:	0	0	1	0	0
$d.2$:	0	0	0	1	0
$d.3$:	0	0	0	0	1

For a Boolean data-word, dual-rail and 1-of-N are obviously identical. For a 2-bit data word, both dual-rail and 1-of-4 codes require four wires. For an N-bit data word, dual-rail requires $2 * N$ wires. If the bits of the original word are paired and each pair is 1-of-4 encoded, this coding also requires $2 * N$ wires. An assignment of a valid value to a dual-rail-coded word requires $2 * N$ transitions, but requires only N transitions in the case of a 1-of-4 code. [See Fig. 6(b).]

F. *k*-out-of-*N* Codes

The 1-of-N code, also called *one-hot*, is a special case of a larger class of codes called *k-out-of-N*. Instead of using just one true bit out of N code bits, as is done in the 1-of-N, we may use k, $0 < k < N$, true bits to represent a valid code value. The number of valid values of a k-out-of-N code is $\binom{N}{k}$. Hence, the maximal number of valid values for a given N is obtained by choosing k as $N/2$. Sperner has proved that this code is not only the optimal k-out-of-N code, but also the optimal DI code in terms of the size of the code set for a given N.

G. Which DI Code?

The choice of a DI code in the design of a system on a chip is dictated by a number of practical requirements. First, the tests for validity and neutrality must be simple. The neutrality test is simple: as in all codes, the unique neutral value is the set of all zeroes or the set of all ones. But the validity test may vary greatly with the code. Second, the coding and decoding of a data word must be simple. Third, the overhead in terms of the number of bits used for a code word compared to the number of bits used for a data word should be kept reasonably small. Finally, the code should be easy to "split": a coded word is often split into portions that are distributed among a number of processes—for example, a processor instruction may be decomposed into an opcode, and several register fields. It is very convenient if the portions of a code word are themselves a valid code word. This is the case for the dual-rail code for all partitions and for the 1-of-4 code for partitions down to a quarter-byte. For all those practical reasons, dual-rail and 1-of-4 are used almost exclusively in asynchronous VLSI design.

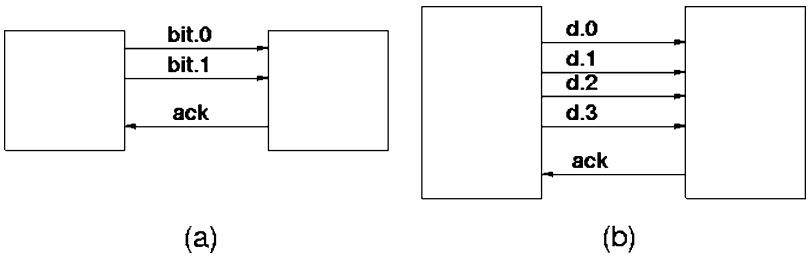

Fig. 6. *(a) A dual-rail coding of a Boolean data-channel. (b) A 1-of-4 coding of a four-valued integer data channel. Observe that no delay element is needed.*

H. Validity and Neutrality Tests

The combination of four-phase handshake protocol and DI code for the data gives the following general implementation for communication on a channel. In this generic description, we use global names for both the sender and receiver variables. A collection of data wires called *data* encodes the message being sent. A single *acknowledge* wire *ack* is used by the receiver to notify the sender that the message has been received. This wire is called the *enable* wire when it is initialized high (true).

The data wires are initially set to the neutral value of the code. The concurrent assignment denoted $data \Uparrow$ takes the data wires from the neutral value to a valid value. The concurrent assignment $data \Downarrow$ takes the data wires from a valid value back to the neutral value. The protocols for sender and receiver can be described as

$$\begin{aligned} &\text{Send}: \quad data \Uparrow;\ [v(ack)];\ data \Downarrow;\ [n(ack)] \\ &\text{Receive}: \quad [v(data)];\ ack \uparrow;\ [n(data)];\ ack \downarrow . \end{aligned}$$

The predicate $v(X)$, called *validity test*, is used to determine that X is a valid value for the chosen DI code. The predicate $n(X)$, called *neutrality test*, is used to determine that x has the neutral value in the chosen DI code. The implementations of validity and neutrality tests play an important role in the efficiency of QDI systems.

1) Active and Passive Protocols: There is an asymmetry in the (two-phase and four-phase) handshake protocols described in the previous section: one side, here the sender, starts by setting some output variables (wires) to a valid value. Such a protocol is called *active*. The other side, here the receiver, starts by waiting for some input variables (wires) to have a valid value. Such a protocol is called *passive*. Symmetrical protocols are possible but are more complicated and therefore rarely used.

Of course, an active protocol on one side of a channel has to be matched to a passive protocol on the other side of the same channel. It seems "natural" to choose the sender side to be active and the receiver side to be passive, but in fact, the sender can be passive and the receiver active. We will see cases when this is a better choice. The protocol is then as follows:

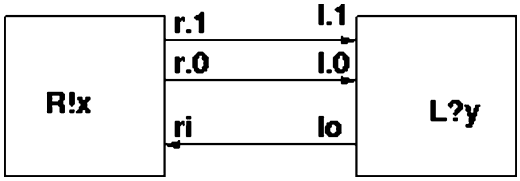

Fig. 7. *Handshake wires for a one-bit DI data channel.*

$$\begin{aligned} &\text{Send}: \quad [v(ack)];\ data \Uparrow;\ [n(ack)];\ data \Downarrow \\ &\text{Receive}: \quad ack \uparrow;\ [v(data)];\ ack \downarrow;\ [n(data)]. \end{aligned}$$

2) Example—One-Bit Channel: A one-bit channel between a sender and a receiver is implemented as in Fig. 7. Two data wires $(r.1, l.1)$ and $(r.0, l.0)$ are used to code the values true and false of the bit. Wire (lo, ri) is often called the acknowledge wire. Next, we implement the send action $R!x$ and the receive action $L?y$, where x is a Boolean variable local to the sender and y is a Boolean variable local to the receiver. The validity and neutrality tests for the receive are $l.1 \vee l.0$ and $\neg l.1 \wedge \neg l.0$. For an active send and a passive receive, we get for $R!x$

$$[x \rightarrow r.1 \uparrow [] \neg x \rightarrow r.0 \uparrow]; [ri]; r.1 \downarrow, r.0 \downarrow; [\neg ri]$$

and for $L?y$

$$[l.1 \vee l.0]; [l.1 \rightarrow y \uparrow [] l.0 \rightarrow y \downarrow]; lo \uparrow; [\neg l.1 \wedge \neg l.0]; lo \downarrow .$$

In the send, the selection $[x \rightarrow r.1 \uparrow [] \neg x \rightarrow r.0 \uparrow]$ assigns the value of x to the data wires of port R. In the receive, the selection $[l.1 \rightarrow y \uparrow [] l.0 \rightarrow y \downarrow]$ assigns the

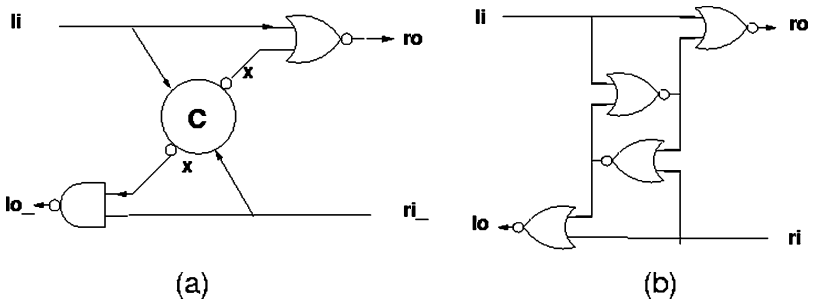

Fig. 8. *Implementation of an active–active buffer (sequencer): (a) with a C-element implementation of the state bit and (b) with a cross-coupled NOR-gate implementation of the state bit. The circle with a C is the symbol for the C-element. It is shown with its inverted output x duplicated.*

value of the data wires of port L to y. In practice, the internal variables x and y are also dual-rail encoded. In the receive, the validity test $[l.1 \vee l.0]$ is superfluous, since the selection following it also includes waiting for $l.1$ or $l.0$. We can rewrite the HSE of the receive $L?y$ as

$$[l.1 \longrightarrow y\uparrow \,[]\, l.0 \rightarrow y\downarrow]; lo\uparrow; [\neg l.1 \wedge \neg l.0]; lo\downarrow .$$

For passive send and active receive, the solution for $R!x$ is

$$[ri]; [x \longrightarrow r.1\uparrow \,[]\, \neg x \rightarrow r.0\uparrow]; [\neg ri]; r.1\downarrow, r.0\downarrow$$

and for $L?y$

$$lo\uparrow; [l.1 \longrightarrow y\uparrow \,[]\, l.0 \longrightarrow y\downarrow]; lo\downarrow; [\neg l.1 \wedge \neg l.0].$$

V. BASIC BUILDING BLOCKS: SEQUENCING, STORAGE, COMPUTATION

The three basic building blocks are: 1) a circuit that sequences two bare communication actions—the sequencing of any two arbitrary actions can be reduced to the sequencing of two bare communications; 2) a circuit that reads and writes a single-bit register; and 3) a circuit that computes a Boolean function of a small number of bits.

A. Sequencer

The basic sequencing building block is the "sequencer" process, also called "left–right buffer" $p1 : *[L; R]$ which repeatedly does a bare communication on its left port L followed by a bare communication on its right port R. The two ports are connected to an *environment* which imposes no restriction on the two communications. The simplest implementation is when both ports are active. (The reason is that a handshake on a passive port is initiated by the environment and therefore requires extra effort to be synchronized.) For L and R (bare) active ports, the HSE of $p1$ is

$$*[lo\uparrow; [li]; \; lo\downarrow; \; [\neg li]; \; ro\uparrow; \; [ri]; \; ro\downarrow; \; [\neg ri]].$$

The state preceding $ro\uparrow$ and the state preceding $lo\uparrow$ are identical in terms of the variables of the HSE, and therefore the states in which each of the two transitions is to fire cannot be separated. We introduce a *state variable* x (initially false) to distinguish those two states

$$*[lo\uparrow; [li]; x\uparrow; lo\downarrow; [\neg li]; ro\uparrow; [ri]; x\downarrow; ro\downarrow; [\neg ri]].$$

Now, all the states that need to be distinguished are uniquely determined and we can generate a PR set that implements the HSE. This leads to the two solutions shown in Fig. 8. In the first solution, the state variable x is implemented with a C-element, in the second one with cross-coupled NOR-gates.

All other forms of the left–right buffer are derived from the active–active buffer by changing an active port into a passive one. The conversion is done by a simple C-element. The passive–active buffer is shown on Fig. 9.

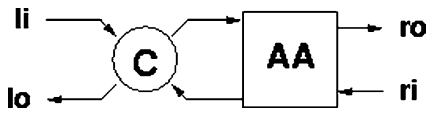

Fig. 9. *A passive–active buffer implemented as an active–active buffer with a C-element as an active-to-passive converter on port L.*

1) Reshuffling and Half-Buffers: We have already mentioned that the down-going phase of a four-phase handshake is solely for the purpose of resetting all variables to their initial (neutral state) values, usually false. The

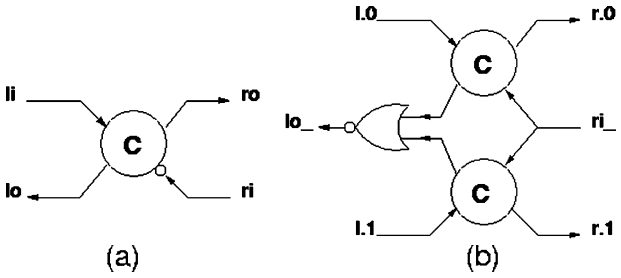

Fig. 10. *A simple half-buffer. (a) Bare handshake. (b) With one bit of data transmitted from L to R.*

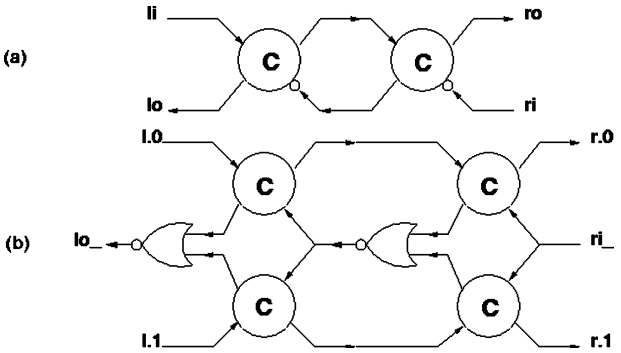

Fig. 11. *A full-buffer FIFO stage. (a) Bare handshake. (b) Transmitting 1 bit of data from left to right.*

designer therefore has some leeway in the sequencing of the down-going actions of a communication with respect to other actions of an HSE. The transformation that moves a part of a handshake sequence in an HSE is called *reshuffling*. It is an important transformation in asynchronous system synthesis as many alternative implementations of the same specification can be understood as being different reshufflings of the same initial HSE. Starting from the HSE of the passive–active buffer

$$*\,[[li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri]]$$

we can apply several reshufflings.

2) Simple Half-Buffer: A rather drastic reshuffling is

$$*\,[[\neg ri];\ [li];\ lo\uparrow;\ ro\uparrow;\ [\neg li];\ [ri];\ lo\downarrow;\ ro\downarrow].$$

Its interest is that it leads to a very simple implementation: a simple C-element with the output replicated to be both *lo* and *ro*, as shown on Fig. 10(a).

By definition, a buffer is such that there is a state in which the number of completed L-communications ($\#L$) exceeds the number of completed R-communications ($\#R$) by one: $\#L = \#R + 1$. A direct implementation of the buffer should have a slack one. But what is the slack of this reshuffling? The reshuffling has decreased the slack between L and R, and therefore there is no longer a state where $\#L = \#R + 1$. But as we shall see momentarily, the sequential composition of two such modules does implement a buffer. Therefore, the C-element implementation is called a *half-buffer* [22], more specifically a *simple half-buffer* (SHB). The SHB is a very useful module to construct simple linear FIFOs. For instance, for L and R Boolean ports, the half-buffer implementation of $*[L?x; R!x]$ is shown in Fig. 10(b). It is not used when computation is involved. The SHB is one of the oldest asynchronous building blocks still in use. It was first introduced by Muller [33].

3) C-Element Full-Buffer: Another (less drastic) reshuffling of the original HSE is

$$*\,[[li];\ lo\uparrow;\ [\neg ri];\ ro\uparrow;\ [\neg li];\ lo\downarrow;\ [ri];\ ro\downarrow]$$

which admits the two-C-element implementation of Fig. 11(a). Since *ri* is false in the neutral state of R, the HS sequence of L can complete without the environment of R being started, i.e., even if *ri* does not change. Hence, the above HSE has a slack one between L and R, and therefore it implements a full-buffer. Since this full-buffer is the linear composition of two simple half-buffers, this explains the term half-buffer used for the previous reshuffling. A full-buffer FIFO stage transmitting one bit of data from L to R is shown in Fig. 11(b).

B. Reshuffling and Slack

Reshuffling is used to simplify implementation. By overlapping two or more handshaking sequences, reshuffling reduces the number of states the system has to step through, often eliminating the need for additional state variables. Reshuffling also makes it possible to pass data directly from an input port—say, L—to an output port—say, R—without using an internal register x. In such a case, we write $R!(L?)$ instead of $L?x; R!x$.

But reshuffling may also reduce the slack of a pipeline stage when it is applied to an input port and an output port, for instance, L and R in the simple buffer. Hence, reshuffling a buffer HSE is usually a tradeoff between reducing the circuit complexity on the one hand, and reducing the slack on the other hand, thereby reducing the throughput.

C. Single-Bit Register

Next, we implement a *register* process that provides read and write access to a single Boolean variable, x. The environment can write a new value into x through port P, and read the current value of x through port Q. Read and

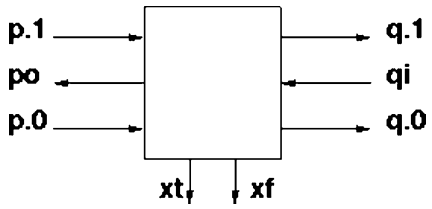

Fig. 12. *Handshake wires for the single-bit register.*

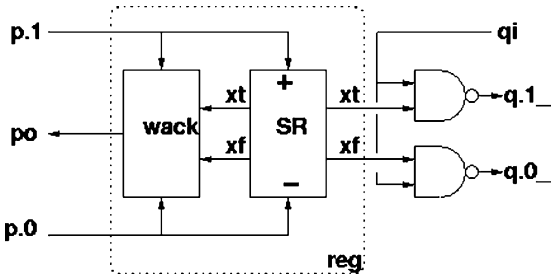

Fig. 13. *An implementation of the single-bit register.*

write requests from the environment are mutually exclusive. As shown in Fig. 12, input port P is implemented with two input wires, $p.1$ for receiving the value **true**, and $p.0$ for receiving the value false; and one acknowledge wire, po. Output port Q is implemented with two output wires, $q.1$ for sending the value **true**, and $q.0$ for sending the value **false**; and one request wire, qi. Variable x is also dual-rail encoded as the pair of variables xt, xf. With passive protocols used for both P and Q, the HSE gives

$$
\begin{aligned}
*[[&p.1 && \longrightarrow xf\downarrow;\ xt\uparrow;\ po\uparrow;\ [\neg p.1];\ po\downarrow\\
\ []&p.0 && \longrightarrow xt\downarrow;\ xf\uparrow;\ po\uparrow;\ [\neg p.0];\ po\downarrow\\
\ []&xt\wedge qi && \longrightarrow q.1\uparrow;\ [\neg qi];\ q.1\downarrow\\
\ []&xf\wedge qi && \longrightarrow q.0\uparrow;\ [\neg qi];\ q.0\downarrow\\
]].&
\end{aligned}
$$

1) Writing an Asynchronous Register: The PRs for the write part of the register (the first two lines of the HSE) are

$$
\begin{aligned}
p.1 &\rightarrow xf\downarrow & \qquad p.0 &\rightarrow xt\downarrow\\
\neg xf &\rightarrow xt\uparrow & \qquad \neg xt &\rightarrow xf\uparrow\\
p.1\wedge xt &\rightarrow po\_\downarrow & \qquad p.0\wedge xf &\rightarrow po\_\downarrow\\
\neg p.1 &\rightarrow po\_\uparrow & \qquad \neg p.0 &\rightarrow po\_\uparrow
\end{aligned}
$$

(We have inverted po as $po\_$ to make it directly implementable in CMOS.) Although it looks straightforward, this PR set and the circuits derived from it deserve scrutiny. The PRs for xt and xf are those of a set–reset gate and can be implemented either with NOR-gates or with inverters (the preferred solution for multibit register files and memories where density is important.) The PRs setting and resetting $po\_$ form what is known as the *write-acknowledge* circuitry or *wack*. They are grouped together as

$$
\begin{aligned}
(p.1\wedge xt)\vee(p.0\wedge xf) &\rightarrow po\_\downarrow\\
\neg p.1\wedge\neg p.0 &\rightarrow po\_\uparrow.
\end{aligned}
$$

A direct CMOS implementation of the above PRs is usually preferred. A pass-transistor implementation is also used when circuit size is important. The *write-acknowledge* (or *wack*) represents the main cost we have to pay for not relying on timing assumptions: since we cannot know how long it takes to set or reset xt and xf, we have to *compute* the information that the writing of xt and xf has completed successfully. In practice, the overhead of *wack* is too high for memories and register-files, and therefore some timing assumptions are usually introduced for density reasons in asynchronous memory design. But write-acknowledge is used in all other QDI circuits.

2) Reading an Asynchronous Register: The read-part of the register is simple. In most cases it can be implemented with the two NAND-gates shown in Fig. 13.

D. N-bit Register and Completion Tree

An n-bit register R is built as the parallel composition of n one-bit registers r_i. Each register r_i produces a single write-acknowledge signal $wack_i$. All the acknowledge signals are combined by an n-input C-element to produce a single write-acknowledge for R. This n-input C-element, say, $y = x1\underline{C}x2\underline{C}\ldots\underline{C}xn$ follows the restricted protocol in which a transition on the output y is always preceded by exactly one transition on each input, as follows:

$$
*[(x1\uparrow, x2\uparrow, \ldots, xn\uparrow); y\uparrow; (x1\downarrow, x2\downarrow, \ldots, xn\downarrow); y\downarrow].
$$

In this case, the n-input C-element can be decomposed into a binary tree of two-input C-elements without unstable transitions on the intermediate variables introduced by the decomposition. Such a C-element tree is called a *completion tree* [29].

The completion tree puts a delay proportional to $\log n$ elementary transitions on the critical cycle. Combined with the write-acknowledge circuit itself, the completion tree constitutes the *completion detection circuit*, which is the main source of inefficiency in QDI design. Numerous efficient implementations of completion detection have been proposed. See in particular [9]. The read part of the n-bit register is straightforward: the read-request signal is

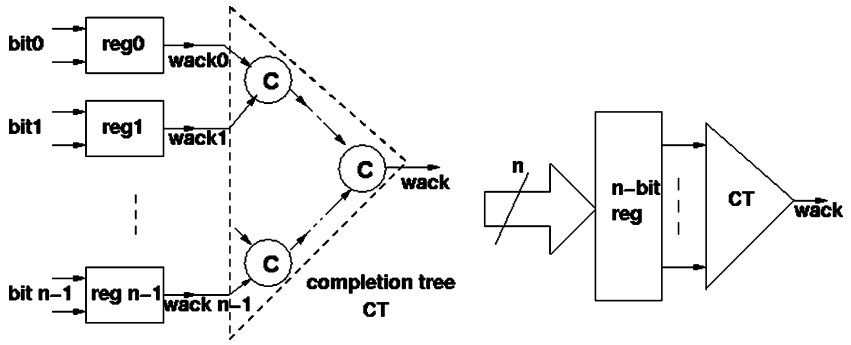

Fig. 14. *An n-bit register as the composition of n single-bit registers. The global write-acknowledge signal is generated by a completion tree combining the single-bit write-acknowledges.*

forked to all bits of the register. The n-bit register is shown in Fig. 14.

E. Completion Trees versus Bundled Data

It is because of completion-tree delays that bundled data is believed by some designers to be more efficient than DI codes for datapath. Completion tree is replaced with a delay line mirroring the delays required to write data into the registers. However, the increasing variability of modern technology requires increasing delay margins for safety. It is the authors' experience that after accounting for all margins, the total delay of bundled data is usually longer than the completion-tree delay—and bundled data gives up the robustness of QDI.

F. Function Evaluation

Consider computing the Boolean function $f(X)$ and assigning the result to Y, with the handshake $F(X, Y)$

$$*[[v(X)]; [f(X) \longrightarrow y.1 \uparrow \; [] \neg f(X) \longrightarrow y.0 \uparrow]; \\ [n(X)]; y.0 \downarrow, y.1 \downarrow].$$

Conditions $v(X)$ and $n(X)$ are the validity and neutrality tests for X; output Y is set to a valid value corresponding to the value of $f(X)$ and then reset to the neutral value. F can be directly implemented as

$$v(X) \wedge f0(X) \rightarrow y.0 \uparrow \qquad n(X) \rightarrow y.0 \downarrow \\ v(X) \wedge f1(X) \rightarrow y.1 \uparrow \qquad n(X) \rightarrow y.1 \downarrow$$

where $f0$ and $f1$ are the coding of $\neg f$ and f, respectively, when X is coded with a dual-rail or 1-of-N code. However, this direct implementation is rarely possible as the neutrality test $n(X)$ requires long chains of p-transistors as shown in the following example.

1) Example: Boolean Equality: The function f is the equality of two Booleans a and b: $[a = b \rightarrow y \uparrow \; [] a \neq b \rightarrow y \downarrow]$. The dual-rail coded version of the function is

$$[(a.0 \wedge b.0) \vee (a.1 \wedge b.1) \longrightarrow y.1 \uparrow \\ [] (a.0 \wedge b.1) \vee (a.1 \wedge b.0) \longrightarrow y.0 \uparrow].$$

In this example, each guard of the dual-rail function evaluation implies the validity of both inputs. Hence, the PRS can be simplified as

$$(a.0 \wedge b.0) \vee (a.1 \wedge b.1) \rightarrow y.1 \uparrow \\ (a.0 \wedge b.1) \vee (a.1 \wedge b.0) \rightarrow y.0 \uparrow \\ \neg a.1 \wedge \neg a.0 \wedge \neg b.1 \wedge \neg b.0 \rightarrow y.1 \downarrow \\ \neg a.1 \wedge \neg a.0 \wedge \neg b.1 \wedge \neg b.0 \rightarrow y.0 \downarrow .$$

Even for this simple function, the neutrality tests (the guards of the last two PRs) require four p-transistors in series.

2) Precharge Function Evaluation: We decouple the validity/neutrality test from the function evaluation in order to simplify the reset condition for the function. In the HSE of F, we introduce a variable v that is assigned the result of the validity and neutrality tests

$$F(X, Y) \equiv *[[v(X)]; v \uparrow; [f1(X) \longrightarrow y.1 \uparrow \; [] f0(X) \longrightarrow y.0 \uparrow]; \\ [n(X)]; v \downarrow; y.0 \downarrow, y.1 \downarrow].$$

The above HSE can be decomposed into the two HSEs

$$\text{VN} \equiv *[[v(X)]; v \uparrow; [n(X)]; v \downarrow], \\ \text{PCF} \equiv *[[v \wedge f1(X) \longrightarrow y.1 \uparrow \; [] v \wedge f0(X) \longrightarrow y.0 \uparrow]; \\ [\neg v]; y.0 \downarrow, y.1 \downarrow].$$

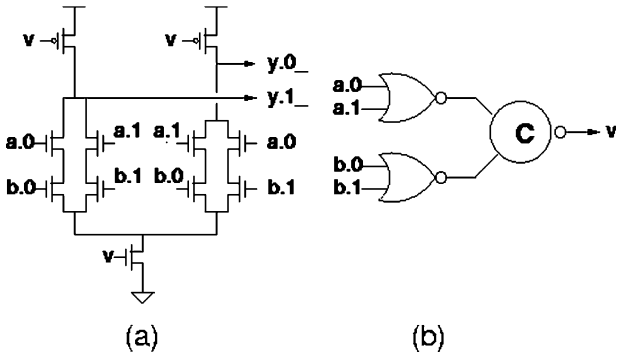

Fig. 15. *Precharge implementation of the Boolean-equality function. (a) Precharge function evaluation (staticizer omitted). (b) Validity/neutrality test circuit. For multiple inputs, the single C-element is replaced with a completion tree.*

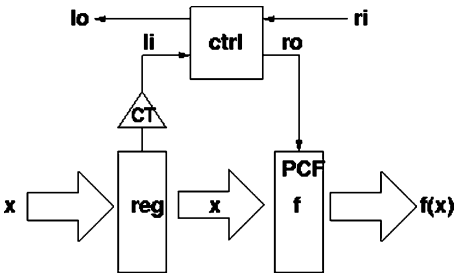

Fig. 16. *The control-data decomposition technique applied to a simple buffer stage. In this case, the input port is active and the output port passive.*

VN computes the validity/neutrality tests. Because of the symmetry of VN, the tests can now be decomposed into a completion tree satisfying the proper limitations on transistor chains. PCF computes the function F as a PCF with v as a control signal. The outputs $y.0\_$ and $y.1\_$ are usually inverted through a staticizer. The circuit implementation of the equality function is shown in Fig. 15.

VI. TWO DESIGN STYLES FOR ASYNCHRONOUS PIPELINES

In systems where throughput is important, computation is usually pipelined. A pipeline stage is a component that receives data on several input ports, computes a function of the data, and sends the result on an output port. The stage may simultaneously compute several functions and send the results on several output ports. Both input and output may be used conditionally. In order to pipeline successive computations of the function, the stage must have slack between input ports and output ports. In this section, we present two different approaches to the design of asynchronous pipelines.

In the first approach, each stage can be complex ("coarse-grain"); the control and datapath of a stage are separated and implemented independently. The decomposition is "syntax-directed." (This style was introduced in [29], and was used in the design of the first asynchronous microprocessor [27].)

The second approach is aimed at fine-grain high-throughput pipelines. The datapath is decomposed into small portions in order to reduce the cost of completion detection, and for each portion, control and datapath are integrated in a single component, usually a precharge half-buffer. The implementation of a pipeline into a collection of fine-grain buffers is based on "data-driven" decomposition [55]. This approach was introduced for the design of the MiniMIPS [28].

A. First Approach: Control-Data Decomposition

In its simplest form, a pipeline stage receives a value x on port L and sends the result of a computation, $f(x)$, on port R. In CHP, it is described as $*[L?x; R!f(x)]$. The design of a pipeline stage combines all three basic operations: sequencing between L and R, storage of parameters, and function evaluation. A simple and systematic approach consists of separating the three functions.

- A control part implements the sequencing between the bare ports of the process, here $*[L; R]$, and provides a slack of one in the pipeline stage.
- A register stores the parameter x received on L.
- A function component computes $f(x)$ and assigns the result to R.

The registers and function components constitute the datapath of the pipeline and are synchronized by the handshake variables of the control. This general scheme is shown in Fig. 16. The control part that implements L as active and R as passive leads to the simplest composition between control and data. If we want to implement the input port L as passive, then the incoming data on L requires extra synchronization until the handshake on L indicates that the register can store the data. This solution is shown in Fig. 17. For the send part (the function evaluation), the implementation is the same whether R is active or passive. If the input port is probed, a special protocol is used that essentially implements the probe as passive and the actual receive as active. The details are omitted.

The above scheme is general and can be applied to any process structure. Given a process, the control is derived by replacing all communication actions with bare communications. The data manipulations—receive, function evaluation, condition evaluation, send—are independent modules that constitute the datapath. Complex conditional expression (guard in selection statements) are also isolated as datapath modules. The modules in the datapath are synchronized by the corresponding handshake signals from the control.

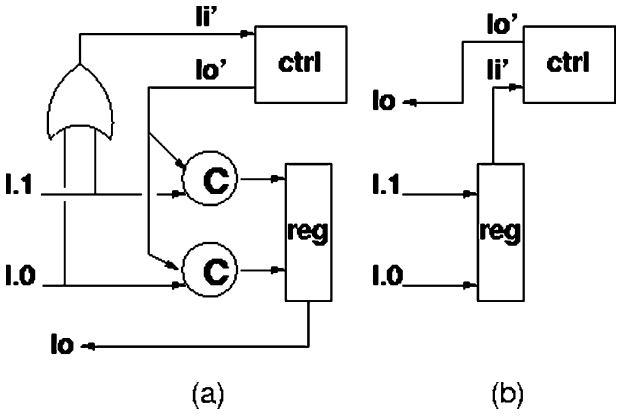

Fig. 17. *Implementation of a one-bit input interface for: (a) a passive port and (b) an active port.*

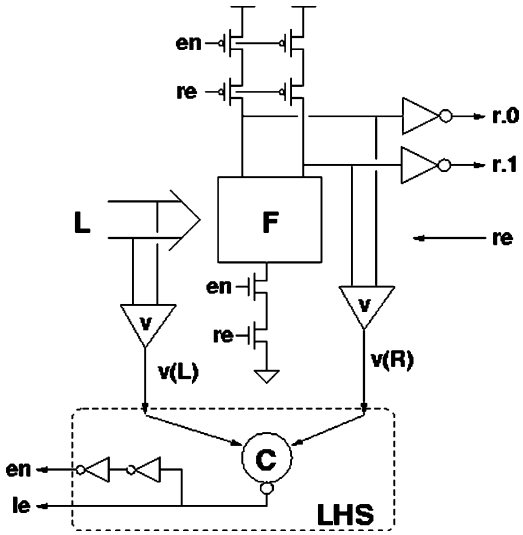

Fig. 18. *Implementation of a simple pipeline stage as a precharge half-buffer. Signal en has been introduced for the sake of generality. In this case, it is identical to le, but not in general.*

B. Second Approach: Integrated Pipelines

Simplicity and generality are the strengths of the previous approach to pipeline design; it allows quick circuit design and synthesis. However, the approach puts high lower bounds on the cycle time, forward latency, and energy per cycle. First, the inputs on L and the outputs on R are not interleaved in the control, putting all eight synchronizing transitions in sequence. Second, the completion-tree delay, which is proportional to the logarithm of the number of bits in the datapath, is included twice in the handshake cycle between two adjacent pipeline stages. Finally, the lack of interleaving in the control handshakes requires the explicit storing of the variable x in a register, adding overhead in terms of both energy and forward latency.

The fine-grain integrated approach we are going to describe next is targeted for high-throughput designs. It eliminates the performance drawbacks of the previous approach by two means: 1) the handshake sequence of L and the handshake sequence of R are reshuffled with respect to each other so as to overlap some of the transitions, and eliminate the need for the explicit registers for input data, and 2) the datapath is decomposed into independent slices so as to reduce the size of the completion trees, and improve the cycle time. In this approach, each slice of the datapath is integrated with its own control to implement a complete computation stage (i.e., combining control and data manipulation) [55].

C. Simple Precharge Half-Buffer (PCHB)

Let us return to the simple pipeline stage $*[L?x; R!f(x)]$ with x Boolean. L is implemented as a passive four-phase handshake, and R as an active four-phase handshake. In the HSE, the acknowledge signals are inverted as *enable* signals *le* and *re* so as to fit better with the inverting logic of CMOS. The PCHB reshuffling eliminates the register variable x by computing the output while the input port still contains the input data. Furthermore, it completes R before completing L

$$*[[re]; [f0(L) \longrightarrow r.0\uparrow \; [] f1(L) \longrightarrow r.1\uparrow]; le\downarrow; \\ [\neg re]; r.0\downarrow, r.1\downarrow; [\neg l.0 \wedge \neg l.1]; le\uparrow].$$

The HSE is decomposed into two components: one, PCF, computing $(r.0, r.1)$ as a standard precharge function block, PCF; the other one, LHS, computing *le*, and in the general case, the internal enable signal *en*

$$\begin{aligned} \text{PCF} &\equiv *[[re \wedge le]; [f0(L) \longrightarrow r.0\uparrow \; [] f1(L) \longrightarrow r.1\uparrow]; \\ &\qquad [\neg re \wedge \neg le]; r.0\downarrow, r.1\downarrow] \\ \text{LHS} &\equiv *[[v(L) \wedge v(R)]; le\downarrow; [\neg v(L) \wedge \neg v(R)]; le\uparrow]. \end{aligned}$$

In LHS, $v(L)$ and $v(R)$ are the validity/neutrality conditions for ports L and R. Because we use only 1-of-n coding for each output, the neutrality condition is the complement of the validity condition, and the test can be implemented with combinational gates only. The production rules for PCF are

$$\begin{aligned} re \wedge le \wedge f0(L) &\rightarrow r.0\uparrow \\ re \wedge le \wedge f1(L) &\rightarrow r.1\uparrow \\ \neg le \wedge \neg re &\rightarrow r.0\downarrow, r.1\downarrow. \end{aligned}$$

The LHS computes the validity/neutrality of inputs and outputs and *le* as $le = v(L)\underline{C}v(R)$. The implementation is shown in Fig. 18. We leave it as an exercise to the reader to

check that the PCHB reshuffling is indeed a half-buffer by checking that the sequential composition of two PCHBs is a full-buffer, i.e., the handshake on L can terminate even when the right environment is blocked. The PCHB has short forward latency—only two elementary transitions, and pipelines composed of PCHBs have excellent throughput (an average of 18 elementary transitions for the MiniMIPS [28]).

D. General PCHB Scheme

For a general PCHB template with multiple input ports and output ports, the implementation is as follows.

1) Each data rail r_j of output port R depending on inputs $L_1, \ldots, L_m$ is the output of a precharge function block

$$r_j = \mathrm{PCF}\big(F_j(L_1, \ldots, L_m), en_R, re\big).$$

where en_R is the internal enable computed by the LHS circuit as follows, and re is the enable rail of R.

2) For each input port L_i, the (inverted) left-enable $l_i.e\_$ is the C-element combination of the validity of L_i and the validity of all outputs R_k that depend on L_i in the current iteration. For unconditional outputs $R_1, R_2, \ldots, R_k$ depending on L_i, we have

$$l_i.e\_ = v(L_i)\underline{C}v(R_1)\underline{C}v(R_2)\ldots\underline{C}v(R_k).$$

The different left-enable computations often share common parts. For instance, if two inputs L_i and L_j are needed by the same group of outputs, the validity of that group of outputs can be shared by le_i and le_j. [See Fig. 19(a).]

3) The internal enable signal en_R is the C-element combination of all left-enable signals $l_i.e$:

$$en_R = l_1.e\underline{C}l_2.e\ldots\underline{C}l_m.e.$$

4) When an input L is used conditionally, and the condition is not provided by a control input, the function block computes the condition as an extra output, say, $c.0, c.1$ where $c.0$ indicates that L is used, and $c.1$ that it is not used. The computation of the left enable le is then done as in Fig. 19(b).

5) Similarly, if a precharge function block does not produce an output for some values of the inputs, a pseudo output is produced for those values of the input so that the validity of the output can be generated in all cases.

E. Split and Merge Components

Controlled split and controlled merge are important network components. They are also examples of PCHB with conditional inputs and conditional outputs. Two solutions are presented, a PCHB implementation and a slack-zero implementation.

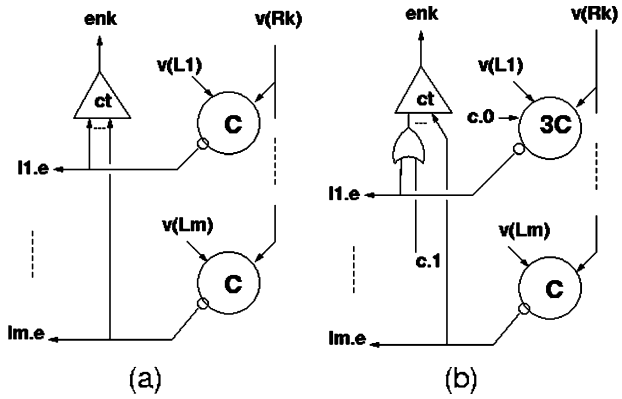

Fig. 19. *Left-enable computation for a general PCHB scheme. (a) LHS when all inputs needed for R_k are unconditional. (b) LHS when one input L_i is conditional. Control signals c.0 ("L_i is used") and c.1 ("L_i is not used") may have to be generated as extra outputs.*

1) Half-Buffer Controlled Merge: A two-way controlled merge merges two input streams from input ports L and M into an output stream on port R. Which port to select for the next input communication is determined by the value received on control port C. The CHP is

$$*\,[C?c; [c.0 \longrightarrow R!(L?)[]c.1 \longrightarrow R!(M?)]].$$

For Boolean ports L, M, and R, the circuit is shown in Fig. 20. The general scheme presented in the previous section has been slightly optimized: the enable signal ce of the control port C can be used as internal enable signal. The computed precharge function is

$$\begin{aligned} ce \wedge re \wedge (c.0 \wedge l.0 \vee c.1 \wedge m.0) &\rightarrow r.0\uparrow \\ ce \wedge re \wedge (c.0 \wedge l.1 \vee c.1 \wedge m.1) &\rightarrow r.1\uparrow \\ \neg ce \wedge \neg re &\rightarrow r.0\downarrow, r.1\downarrow. \end{aligned}$$

The left enables and internal enable are computed as

$$\begin{aligned} le\_ &= v(L)\underline{C}v(R)\underline{C}c.0 \\ me\_ &= v(M)\underline{C}v(R)\underline{C}c.1 \\ ce &= le \vee me. \end{aligned}$$

2) Half-Buffer Controlled Split: The two-way controlled split receives data on one input port L and sends it on one of two output ports M and R. Which output port is selected

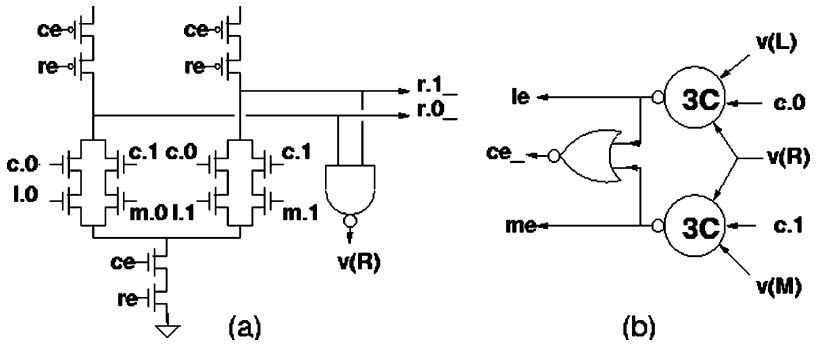

Fig. 20. *A two-way controlled merge implemented as a PCHB. (a) Precharge function. (b) LHS.*

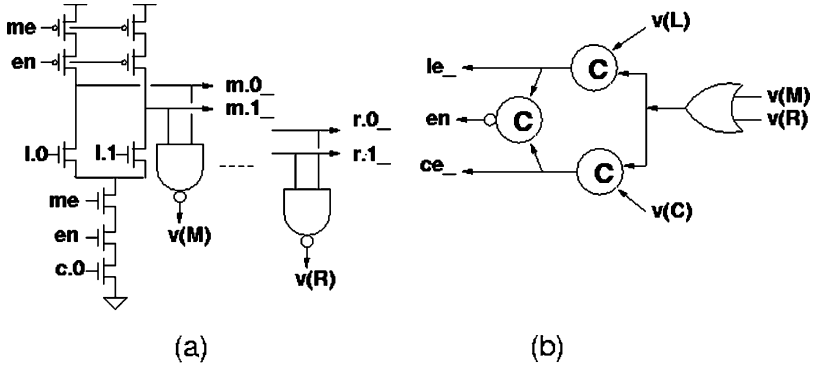

Fig. 21. *A two-way controlled split implemented as a PCHB. (a) Precharge function block for M. (b) LHS.*

is determined by the value received on control port C. The CHP is

$$*\,[C?c;\,[c.0 \longrightarrow M!(L?)[]c.1 \longrightarrow R!(L?)]].$$

For Boolean ports L, M, and R, the circuit is shown in Fig. 21. This is a simple case of conditional output, since exactly one of the two outputs is always used. The computed precharge function is

$$\begin{aligned} me \wedge en \wedge c.0 \wedge l.0 &\rightarrow m.0\uparrow \\ me \wedge en \wedge c.1 \wedge l.0 &\rightarrow r.0\uparrow \\ re \wedge en \wedge c.0 \wedge l.1 &\rightarrow m.1\uparrow \\ re \wedge en \wedge c.1 \wedge l.1 &\rightarrow r.1\uparrow \\ \neg re \wedge \neg en &\rightarrow r.0\downarrow, r.1\downarrow \\ \neg me \wedge \neg en &\rightarrow m.0\downarrow, m.1\downarrow . \end{aligned}$$

The equations for LHS are as follows:

$$\begin{aligned} le\_ &= v(L)\underline{C}(v(R) \vee v(M)) \\ ce\_ &= v(C)\underline{C}(v(R) \vee v(M)) \\ en &= le\underline{C}ce. \end{aligned}$$

3) Example—Two Streams Sharing a Channel: Large data channels are often a scarce resource in an SoC, and mechanisms to share them are important. Let us first look at the simple case when a channel C is shared between two streams. More precisely, we want to establish a channel between send port A and receive port A' using C, or between send port B and receive B' using C. When C is used for a communication between A and A', it should not be used for a communication between B and B', and *vice versa*.

The simplest case is when the exclusive use of the channel is controlled centrally: a control signal is sent both to the controlled-merge process merging A and B into C, and to the controlled-split process forking C to A' and B'. The control signal determines which of the two streams uses the channel for the next communication as in Fig. 22.

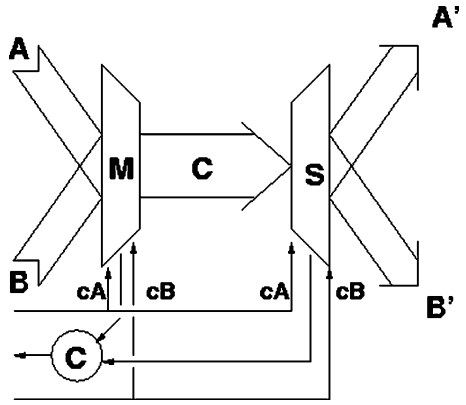

Fig. 22. *Two streams sharing channel C under control of dual-rail signal cA, cB.*

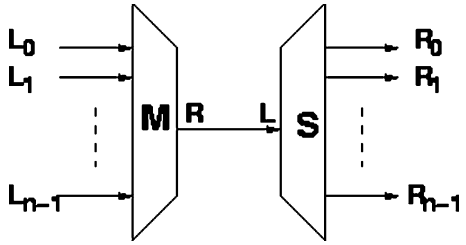

Fig. 23. *A many-to-many bus composed of a many-to-one merge and a one-to-many split.*

VII. ASYNCHRONOUS BUSES

The previous case is a simple version of a bus. A bus is a many-to-one merge of data streams, or a one-to-many split of data streams, or a combination of both. (See Fig. 23.) In a microprocessor, for example, buses are used to send the parameters of an instruction from the register file to the different execution units, or to send the results of an instruction execution from an execution unit to the register file. In that case, the control inputs to the merge and split components of the bus are produced by the instruction decoder.

There are many implementations of asynchronous buses. We show one based on the PCHB. The solution brings to light an annoying problem in asynchronous design: the efficient CMOS implementation of an n-input nor gate when n is large. All solutions we know for the merge/split design contain at least one n-input NOR-gate, where n is either the number of merge inputs or the number of split outputs.

Direct implementation is impossible because of the CMOS restriction on the length of p-transistor pull-up chains. Distributed implementations as trees of two-input OR-gates is possible without hazard because only one input is exercised at a time, but it seriously taxes the throughput of the bus.

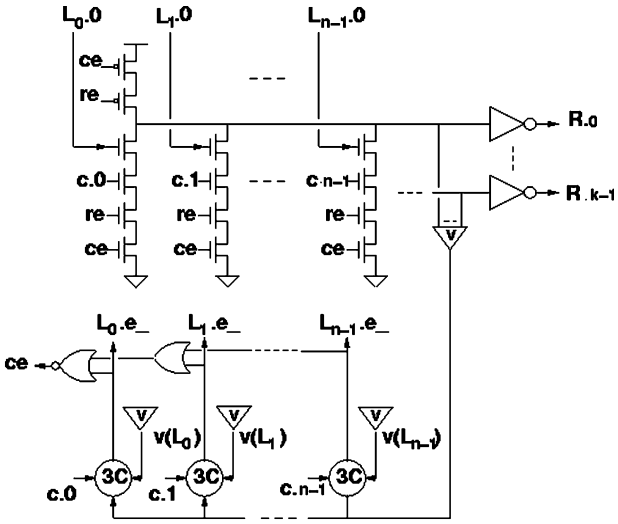

Fig. 24. *A PCHB implementation of a many-to-one bus. The triangles marked with a letter v are combinational gates computing the validity test of input ports L_0 through L_{n-1} and single output port R with bit lines R.0 through R.k − 1. Observe that port C does not need an explicit validity test.*

A. PCHB Implementation of a Many-to-One Bus

The PCHB implementation of a many-to-one bus is a straightforward extension of the two-way merge. The bus has n data input ports L_0 through L_{n-1}, a 1-of-N control input C used to select an input port, and one data output port R. The equations for the left-enables $l_k.e$ for k from 0 to $n-1$, and for the internal enable are as follows:

$$l_k.e\_ = v(L_k)\underline{C}v(R)\underline{C}c.k$$
$$ce\_ = \left(\bigvee k : 0..n-1 : l_k.e\right)$$
$$en = ce.$$

The implementation is shown in Fig. 24.

B. PCHB Implementation of a One-to-Many Bus

The PCHB implementation of a one-to-many bus is a straightforward extension of the two-way split. The bus has n data output ports R_0 through R_{n-1}, a 1-of-N control input C used to select an output port, and one data input port L. We choose the following implementation for the left-enables and internal enable (others are possible):

$$ce\_ = \left(\bigvee k : 0..n-1 : v(R_k)\underline{C}c.k\right)$$
$$le\_ = ce\_\underline{C}v(L)$$
$$en = le.$$

The implementation is shown in Fig. 25.

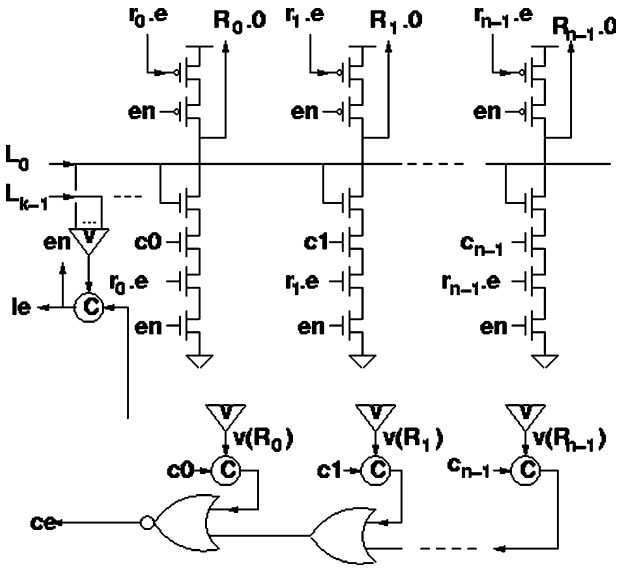

Fig. 25. *A PCHB implementation of a one-to-many bus. The triangles marked with a letter v are combinational gates computing the validity test of input port L with bit lines L.0 through L.k − 1 and output ports R_0 through R_{n-1}.*

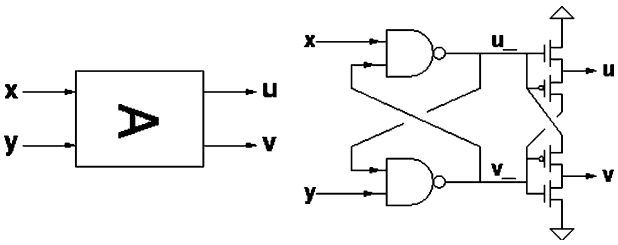

Fig. 26. *(a) The simple arbiter or mutual-exclusion element selects between two possibly concurrent inputs x and y and produces mutually exclusive outputs u and v. (b) An implementation of the basic arbiter consisting of two cross-coupled NAND-gates and a filter. The filter eliminates the spurious values of the NAND-gates outputs produced during the metastable state.*

VIII. ARBITRATION

All circuits presented in the previous sections have stable and noninterfering PR sets. But in order to implement selections in which the conditions are *not* mutually exclusive (nondeterministic choice), for instance, to select between an external interrupt signal and the internal "next-instruction" signal in a microprocessor, or for synchronization between the clocked components of a GALS system, at least one operator must provide a nondeterministic choice between two true guards.

Since stability and noninterference are equivalent to determinism, a nondeterministic computation does not admit stable and noninterfering circuit implementation. Therefore, any implementation of a nondeterministic computation contains nonmonotonic transitions that require special nondigital solutions. Those special circuits are encapsulated inside two primitive building blocks: the *arbiter* and the *synchronizer*.

A fundamental, and long misunderstood, issue related to implementing a nondeterministic choice is that of *metastability*. In classical physics, it is impossible to put an upper bound on the time it takes for a device to make a nondeterministic decision between two alternatives. When the system starts in a state where the physical parameters are such that either alternative can be selected, the device may enter a metastable state in which it may stay an arbitrary length of time before deciding one way or the other [6].

A. Basic Arbiter

The simplest device to make a selection between nonexclusive guards is the *basic arbiter*, or *mutual exclusion element*. Its HSE specification is

$$arb \equiv * [[x \longrightarrow u\uparrow;\ [\neg x];\ u\downarrow \ |\ y \longrightarrow v\uparrow;\ [\neg y];\ v\downarrow]]$$

where x and y are simple Boolean variables. (The "thin bar" | indicates that the two guards can be true at the same time and thus that arbitration between the guards is needed.) The arbiter is usually represented as in Fig. 26(a). Initially, $\neg u \wedge \neg v$ holds. When either x or y or both become true, either u or v is raised but not both. After u is raised the environment resets x to false, and similarly if v is raised. After x has been observed to be low, u is lowered; and similarly if v was raised. Hence, if $\neg u \wedge \neg v$ holds initially, $\neg u \vee \neg v$ holds at any time.

The proper operation of the arbiter requires that two inputs be stable, i.e., once x or y has been evaluated to true, it remains true at least until an acknowledgment transition takes place. If one of the requests is withdrawn before the arbiter has produced an output, the arbiter may fail: one or both outputs may glitch.

B. Implementation and Metastability

Let us first consider the PR sets for *arb* that contain unstable rules. The PR set for the "unstable arbiter" (in which u and v have been replaced with their inverses $u\_$ and $v\_$) is as follows:

$$x \wedge v\_ \rightarrow u\_\downarrow$$
$$y \wedge u\_ \rightarrow v\_\downarrow$$
$$\neg x \vee \neg v\_ \rightarrow u\_\uparrow$$
$$\neg y \vee \neg u\_ \rightarrow v\_\uparrow .$$

The first two PRs of the arbiter are unstable and can fire concurrently. In the digital domain, when started in the state with $x \wedge y$ and $\neg u\_ \wedge \neg v\_$, the set of PRs specifying the arbiter may produce the unbounded sequence of firings: $*[(u\_\downarrow, v\_\downarrow); (u\_\uparrow, v\_\uparrow)]$.

In the analog domain, the state of the arbiter in which x and y are true and the voltages of both $u\_$ and $v\_$ are halfway between Vdd and ground is called *metastable*. Nodes $u\_$ and $v\_$ may oscillate and then stabilize to a common intermediate voltage value for an unbounded period. Eventually, the inherent asymmetry of the physical realization (impurities, fabrication flaws, thermal noise, etc.) will force the system into one of the two stable states where $u\_ \neq v\_$. But there is no upper bound on the time the metastable state will last, which means that it is impossible to include an arbitration device into a clocked system with absolute certainty that a timing failure cannot occur.

In order to eliminate the spurious values of $u\_$ and $v\_$ produced during the metastable state, we compose the "bare" arbiter with a *filter* taking $u\_$ and $v\_$ as input and producing u and v as "filtered outputs." (An nMOS implementation of filter is shown in [47]. The complete nMOS circuit for the arbiter is described in [45]. A CMOS version appeared first in [25].)

(In the CMOS construction of the filter shown in Fig. 26(b), we use the threshold voltages to our advantage: the channel of transistor $t1$ is conducting only when $(\neg u\_ \wedge v\_)$ holds, and the channel of transistor $t2$ is conducting only when $(\neg v\_ \wedge u\_)$ holds.) In QDI design, the correct functioning of a circuit containing an arbiter is independent of the duration of the metastable state; therefore, relatively simple implementations of arbiters can be used. In synchronous design, however, the implementations have to meet the additional constraint that the probability of the metastable state lasting longer than the clock period should be negligible.

C. Channel Arbiter

If a process has to arbitrate between two nonmutually exclusive channels, A and B, the channel arbiter CARB serves as an interface to provide mutually exclusive channels A' and B'

$$*[[\overline{A} \longrightarrow A'; A | \overline{B} \longrightarrow B'; B]].$$

Since A and B are probed, they are implemented with a passive handshake, and A' and B' with an active handshake. The circuit is shown in Fig. 27. The previous design can be used to arbitrate between channels with data. The arbiter takes as inputs a signal from each data channel A and B indicating that the channel has valid data. If a dual-rail code is used, it suffices to look at the validity of a single bit (a simple OR-gate of the rails of the bit). C-elements prevent the propagation of the data until the channel has been selected by the arbiter.

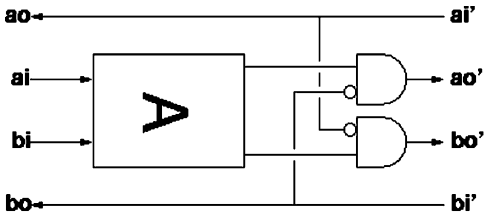

Fig. 27. *A slack-zero channel arbiter. The NAND-gates prevent the second of two requests from proceeding before the HS of the first one is completed.*

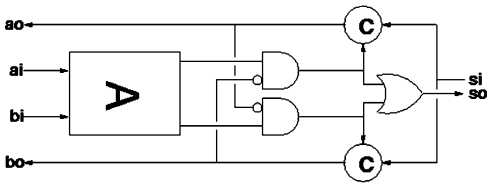

Fig. 28. *A multiplexed arbiter XARB composed of a channel arbiter and a simple multiplexer.*

D. Multiplexed Arbitration

A useful building block, the multiplexed arbiter XARB, is an extension of the channel arbiter with a multiplexer (merge)

$$*[[\overline{A} \longrightarrow S; A | \overline{B} \longrightarrow S; B]].$$

It is obtained by combining the channel arbiter CARB with a simple multiplexer (merge). The multiplexed arbiter XARB is shown in Fig. 28.

E. Multichannel Tree Arbiter

Thanks to the multiplexed arbiter XARB, we can generalize the simple channel arbiter to the case of more than two channels. The implementation of four-channel arbiter ARB4 as an arbiter tree is shown in Fig. 29. The generalization to an arbitrary number of channels is straightforward. The tree need not be balanced, and can be incomplete if the number of channels is not a power of two.

F. Synchronizer

As we argued earlier, the proper operation of the arbiter requires that the inputs x and y be stable, i.e., once they are true, they remain asserted until the arbiter has raised one of the corresponding outputs. Therefore, we cannot use an arbiter to sample a completely unsynchronized external signal, like an interrupt signal or other

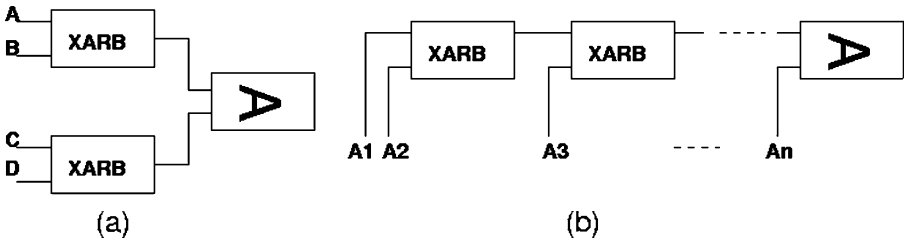

Fig. 29. *A four-way tree arbiter composed of two multiplexed arbiters and a bare arbiter as the root. The generalization to an arbitrary number of channels is straightforward. The tree need not be balanced, and can be incomplete if the number of channels is not a power of two.*

peripheral signal. A circuit to solve this problem is called a *synchronizer*. The synchronizer has two inputs: a control input re and the input signal x that is to be sampled. We have chosen to specify it in such a way that it produces a dual-rail output $(r.0, r.1)$ with the two rails representing the true and false values of the sampled input x. The HSE specification of the synchronizer is

$$\begin{aligned}\text{sync} \equiv *[&[re \wedge \neg x \longrightarrow r.0\uparrow;\ [\neg re]; r.0\downarrow\\ &|re \wedge x \longrightarrow r.1\uparrow;\ [\neg re]; r.1\downarrow]].\end{aligned}$$

When the environment raises re, the synchronizer starts evaluating x and returns the observed value by raising either $r.0$ or $r.1$. The difficulty is that x may change its value at any time and therefore also during its evaluation by the synchronizer. (That is the reason why both guards can evaluate to true, which explains the use of the thin bar.) If x has a stable true value within a finite, bounded interval around the time re is raised, the synchronizer asserts $r.1$, and similarly if x is a stable false; otherwise, the circuit asserts either $r.1$ or $r.0$, but not both. In practice, the "confusion interval" is a very short period, approximately the delay of the single inverter used to invert the input. But the synchronizer must work correctly—i.e., raise either $r.1$ or $r.0$—even when x changes during the confusion interval. Implementing a synchronizer properly is difficult enough that it is usually avoided.

Unlike the arbiter, the synchronizer as defined by sync cannot be implemented directly by feeding the two inputs into a bistable device (cross-coupled NAND gates). To see why, consider that the program has to do two things to advance from waiting for $re \wedge x$ to $r.0\uparrow$: first, the second guard must be "locked out," so that $r.1$ cannot happen; second, the assignment $r.0\uparrow$ must take place. But if x should change after the second guard has been locked out but before the first guard has been selected, the program will deadlock. There is a race condition due to the fact that the two guards can be invalidated at any time. Removing the race condition requires introducing intermediate *stable* variables $a.0$ and $a.1$ (they remain true once evaluated to true) to stabilize the selection of one of the two guards, leading to HSE sync1

$$\begin{aligned}*[&[re \wedge \neg x \rightarrow a.0\uparrow; [a.0]; r.0\uparrow; [\neg re]; a.0\downarrow; [\neg a.0]; r.0\downarrow\\ &|re \wedge x \rightarrow a.1\uparrow; [a.1]; r.1\uparrow; [\neg re]; a.1\downarrow; [\neg a.1]; r.1\downarrow]].\end{aligned}$$

We decompose sync1 into three parts: int0 and int1 "integrate" the possibly oscillating values of x into a stable false value $a.0$ and a stable true value $a.1$, and SEL selects between $a.0$ and $a.1$

$$\begin{aligned}\text{int0} &\equiv *[[re \wedge \neg x \longrightarrow a.0\uparrow;\ [\neg re];\ a.0\downarrow]],\\ \text{int1} &\equiv *[[re \wedge x \longrightarrow a.1\uparrow;\ [\neg re];\ a.1\downarrow]],\\ \text{SEL} &\equiv *[[a.0 \longrightarrow r.0\uparrow;\ [\neg a.0 \wedge \neg a.1];\ r.0\downarrow\\ &\qquad |a.1 \longrightarrow r.1\uparrow;\ [\neg a.1 \wedge \neg a.0];\ r.1\downarrow]].\end{aligned}$$

The parallel composition of int0, int1, and SEL is not strictly equivalent to sync1 because, in the decomposition, the control can advance simultaneously to $a.0\uparrow$ and $a.1\uparrow$, which is not possible in the original sync1. Consequently, the arbitration takes place between $a.0$ and $a.1$ in SEL. But now the arbitration is between stable signals. However, SEL is not exactly an arbiter: in an arbiter, when both inputs are asserted, the arbiter selects both of them in an arbitrary order, since a request is never withdrawn. In SEL, when both inputs are asserted, only one should be selected: whichever is chosen to be the value of the input when it is sampled. Hence, SEL must check that both $a.0$ and $a.1$ are false before resetting the output $r.0$ or $r.1$.

Nevertheless SEL can be implemented directly as a bistable device in the same way as the arbiter. The production rules for the bistable component are

$$\begin{aligned}r.1\_ \wedge a.0 &\rightarrow r.0\_\downarrow\\ \neg r.1\_ \vee (\neg a.0 \wedge \neg a.1) &\rightarrow r.0\_\uparrow\\ r.0\_ \wedge a.1 &\rightarrow r.1\_\downarrow\\ \neg r.0\_ \vee (\neg a.1 \wedge \neg a.0) &\rightarrow r.1\_\uparrow .\end{aligned}$$

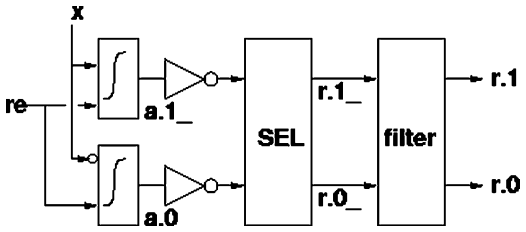

Fig. 30. *A synchronizer circuit consisting of two integrators (producing stable copies a.1 and a.0 of x and its inverse), a bistable device, and a filter.*

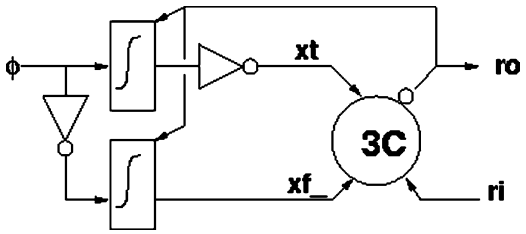

Fig. 31. *A conceptual implementation of the clock-handshake interface. The two integrators produce stable copies of the clock and its inverse; the 3-input C-element implements the four-phase handshake and produces the control input for the integrators.*

The complete circuit for the synchronizer comprising the two integrators, the bistable device and the metastability filter is shown in Fig. 30. This is only one in a family of solutions [39]. A completely different solution is proposed in [43].

G. A Clock/Handshake Interface

The following case study is a remarkable circuit that interfaces a clock signal with a handshake protocol. More precisely, the circuit transforms a clock signal ϕ into a handshake protocol between variables *ro* and *ri*. The HSE of the clock/handshake interface (CHI) is

$$*\,[[\neg\phi];\ ro\uparrow;\ [ri];\ [\phi];\ ro\downarrow;\ [\neg ri]].$$

If we introduce two integrators to generate stable copies *xt* and *xf* of the true and false values of ϕ, the HSE becomes

$$*\,[[\neg ri \wedge \neg xt \wedge \neg xf\_];\ ro\uparrow;\ [ri \wedge xt \wedge xf\_];\ ro\downarrow].$$

which is just a three-input C-element. A conceptual implementation of CHI is shown in Fig. 31.

The integrators may not work correctly for any input waveform on ϕ. For instance, a very slow rising *xt* could trigger the C-element well before the voltage on *xt* reaches *Vdd*. In this case, we should have to depend on a timing assumption to guarantee that *xt* reaches *Vdd* before *xf*_ switches; otherwise, the intermediate voltage could be interpreted as false by the C-element later. This potential analog problem can be removed by adding inverters on the *xt* and *xf* outputs implemented as Schmitt triggers. We are using the property of the Schmitt trigger that if the input changes *monotonically* from one logic value to the other, the output switches *quickly* from one value to the other. Since the integrators see to it that the outputs change monotonically regardless of the inputs, the Schmitt triggers guarantee that *xt* and *xf* switch quickly enough that neither node is observed as both true and false. This (practical) implementation is shown in Fig. 32.

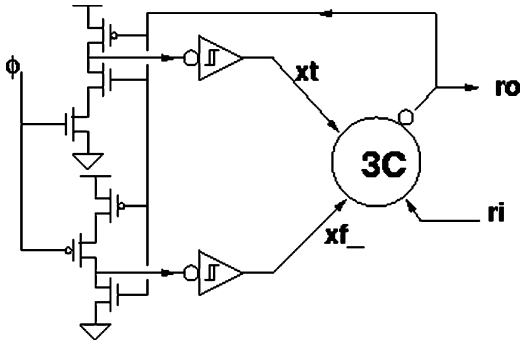

Fig. 32. *A practical implementation of the clock-handshake interface. The outputs of the integrators have been augmented with Schmitt-trigger inverters to guarantee quick transitions on xt and xf, and the inverter on the input of the bottom integrator has been eliminated.*

IX. GALS AND ASYNCHRONOUS-TO-SYNCHRONOUS CONVERSION

Historically, asynchronous communication between synchronous components was prevalent in computer systems. For instance, the UNIBUS used in DEC PDP-11 and VAX computers was a completely asynchronous bus [50], [51]; to this day, the common SCSI protocol for peripheral communications supports handshake-based asynchronous signalling [54]. In the case of our concrete problem, the situation is one where *P* communicates with *Q* through an asynchronous middleman *R* (e.g., the UNIBUS). The transfer of data commences with *P*'s sending the data to *R*, whence it proceeds to *Q*. The interface between *P* and *R* as well as that between *R* and *Q* are both examples of an asynchronous-synchronous (AS) crossing, and have similar solutions. Because of the direction of data movement in the example, the $R-Q$ interface is somewhat more complicated, which is why we will discuss it; the $P-R$ interface can be inferred through simplification. In all that follows, except the section that specifically deals with multiple inputs, we will be analyzing the simple case

where there is one "asynchronous part" that is sending data to another "synchronous part."

The reason that the AS problem has confounded so many system designers is simply this: when we are given to synchronize asynchronous signals into a synchronous environment with a free-running periodic clock, there is an *unavoidable* tradeoff between data communications latency and a probability of *catastrophic system failure*. If the communications latency is reduced, then the probability of catastrophic system failure increases, and if the probability of catastrophic system failure is to be reduced, we must yield some communications latency. This cannot be avoided. Similarly, no matter how much communications latency we are willing to live with, some small degree of catastrophic system failure remains. The problem *cannot* be avoided with less drastic means than introducing a scheme for stopping the clock (see below) or giving up synchronous operation entirely (see the preceding sections of this paper).

The choice of the word "catastrophic" is deliberate. The behavior of a synchronous system that has undergone synchronization failure cannot be usefully bounded, except probabilistically. Sometimes one can read in the literature of some scheme where the inventor claims that he can build a system that synchronizes an asynchronous signal to a clock with a known, finite latency, and either: 1) there is no possibility of synchronization failure or 2) if synchronization failure does occur, it will only take some definite, benign form. Both alternatives are impossible, but it sometimes takes some careful analysis to show why a particular circuit fails to achieve them; not rarely is it the case that while the circuit in question may not suffer disastrous internal consequences due to a synchronization failure, but instead its output signals are not guaranteed to satisfy setup and hold times imposed by the environment. Such circuits are frequently useful, as they may increase the amount of time available for metastability resolution, but nevertheless, they cannot ever be completely reliable.

To repeat, all schemes that aim at finite-latency communication with zero probability of catastrophic system failure are doomed. Therefore, let us state as a design principle at the outset that as part of the design of any scheme for AS conversion, the designer must identify, and ideally evaluate the probability of, the scenario in which the system will fail catastrophically; as the problem cannot be avoided, it does no good to sweep it under the rug.

1) Asynchronous-Synchronous Interface Using Two-Phase Nonoverlapping Clocks: Consider a synchronous system that operates off two-phase (ϕ_0 and ϕ_1 and their inverses) nonoverlapping clocks. This is the standard method of designing custom CMOS circuits and is treated in detail by Mead and Conway [31] and Glasser and Dobberpuhl [56]. By means that do not concern us at the moment, two phases of the clock are generated such that the predicate $\neg\phi_{0,i} \vee \neg\phi_{1,j}$ holds in all locations i, j of the system where the clock phases are available. In other words, it is never the case that both clock phases are high, even taking clock skew into account. Normally the clock generator is built on chip, but it is also possible to put it off chip, if we can tolerate the extra pins; the advantage of putting the clock generator off chip is that we can adjust the nonoverlap time between the phases to compensate for unexpected skews in the manufactured part.

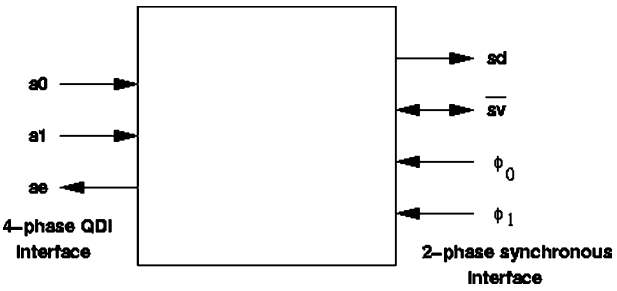

Fig. 33. *Block diagram of interface between four-phase QDI and two-phase synchronous pulse handshakes.*

While it is generally better to use four-phase handshakes in asynchronous systems because of the simplicity of implementation, the situation is not as clear in synchronous systems. We will study the data transfer between a four-phase QDI system and a standard single-rail synchronous system, and in this case, the synchronous side of the interface can more easily be implemented with a two-phase handshake because we do not need to keep track of the phase of the data. In fact, because of the timing guarantees that it is possible to enforce using the clock, we can use the two-phase "pulse handshake" used in APL circuits (Section II)

$$*\,[[\neg d0 \wedge \neg d1];\ d0\uparrow \mid d1\uparrow]\,\|\,*\,[[d0 \vee d1];\ d0\downarrow, d1\downarrow].$$

(We use the vertical thin bar | to denote nondeterministic choice.) This handshake has the speed advantage of a two-phase handshake and the circuit advantage of a four-phase handshake that the data sense is always positive. A block diagram appears in Fig. 33. The variables $a0$, $a1$, and ae represent a standard QDI bit-wide channel (the generalization of the circuit to a wider datapath is obvious). The variables *sd* and *sv_* represent data and validity (inverted for convenience) for a single-rail handshake channel, and of course ϕ_0 and ϕ_1 represent the two clock phases. Fig. 34 shows how we would use the interface in a standard clocked system (logic between the ϕ_1 latch and the ϕ_0 latch is not shown but could be present).

As long as *sv_* obeys the synchronous timing discipline, i.e., as long as *sv_* is stable or falls to zero during ϕ_1, the synchronous access to the data value *sd* presents no

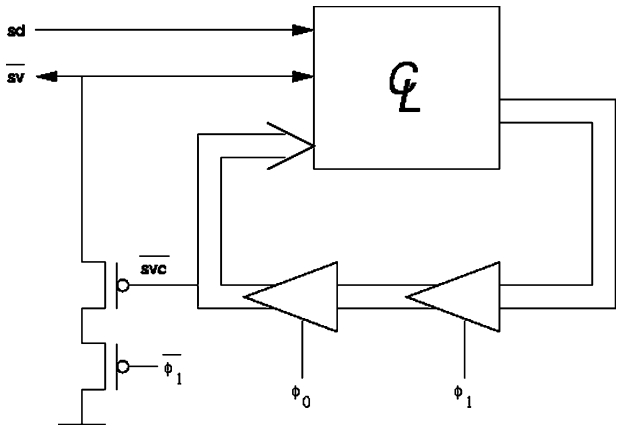

Fig. 34. *Connecting synchronous circuit with two-phase nonoverlapping clock to the asynchronous interface; logic between ϕ_1 and ϕ_0 latches not shown.*

problem. It is simply: wait for $sv\_$ to become asserted (low), operate on sd as necessary, and clear $sv\_$ (set it high). Therefore, any synchronization problem will involve $sv\_$, which justifies our decision to investigate a single-bit circuit. The interface implements the simple CHP program $*[A?d; S!d]$.

Let us proceed with developing the asynchronous controller. The handshakes are a four-phase QDI handshake interleaved with a two-phase pulse handshake, where we shall insert waits for the clock at strategic times, so as to remove any unnecessary race conditions from the circuit. The HSE is as follows:

$$*[[sv\_]; ae\uparrow; [a0 \longrightarrow sd\downarrow \,[]\, a1 \longrightarrow sd\uparrow]; [\phi_0]; sv\_\downarrow; ae\downarrow; [\neg a0 \wedge \neg a1]].$$

Here we have inserted a wait for ϕ_0 before $sv\_\downarrow$; comparing this to the circuit shown in Fig. 33, we can see that this wait can guarantee noninterference on $sv\_$.

Because of the action of the clock, the wait $[\phi_0]$ may be unstable; it is exactly the effect of this instability on the synchronous side of the interface that can cause synchronization failure. How this happens will be obvious once we examine the PRS compilation, which is as follows:

$$\begin{aligned} \neg a0 \wedge a1 \wedge sv\_ &\rightarrow ae\uparrow \\ a0 &\rightarrow sd\downarrow \\ a1 &\rightarrow sd\uparrow \\ \phi_0 \wedge (a1 \wedge sd \vee a0 \wedge \neg sd) &\rightarrow sv\_\downarrow \\ (a0 \vee a1) \wedge \neg sv\_ &\rightarrow ae\downarrow . \end{aligned}$$

And to this we can add the implementation of the synchronous reset of $sv\_$

$$\neg svc\_ \wedge \neg \phi_1\_ \rightarrow sv\_\uparrow .$$

Variable $svc\_$ is internal to the synchronous part.

The production rules for ae can be split into an NOR-gate and a two-input C-element; with this implementation, the generalization of the circuit to wider datapaths becomes obvious. However, the negation of sd in the rule for $sv\_\downarrow$ and the use of the nonnegated $a1$ in the rule for $sd\uparrow$ are more interesting. In ordinary QDI circuits, we would not permit these constructs, as they imply the presence of inverters with unacknowledged output transitions. In this case, however, these transitions are "acknowledged" by the clock—if the circuit turns out not to operate at a high clock speed due to these unacknowledged transitions, slowing it down will fix the problem.

Let us then return to the synchronization failure scenario. Synchronization failure occurs only when the instability of the guard $\phi_0 \wedge (a1 \wedge sd \vee a0 \wedge \neg sd)$ manifests itself; that is, when $sv\_$ is falling and ϕ_0 falls simultaneously, cutting off the transition prematurely. If we examine a straightforward transistor-level implementation of the circuit (Fig. 35), we see that this scenario will leave $sv\_$ at an indeterminate level between a logic zero and a logic one. The action of the staticizer will eventually drive the value of $sv\_$ to a proper zero or one, but this system has a metastable state, which can persist for an arbitrarily long time. The situation is depicted graphically in the timing diagram in Fig. 36. If the outputs, say, x and y, of the combinational logic block have not reached legal logic levels by the time ϕ_1 falls or if their values are inconsistent (i.e., the value of x implies that $sv\_$ was false and that of y implies that $sv\_$ was true), then the system has failed, quite possibly in a catastrophic, unrecoverable way. As usual, the probability of failure can be calculated based on the resolution time of the staticizer on $sv\_$ (marked S in Fig. 35). The probability of failure can be reduced by adding clocked latches on the connection of $sv\_$ with the combinational logic, thereby increasing the time available for resolution (as well as the communications latency).

Two remarks about the staticizer S are in order. First, Dally reports that one of the most common errors in synchronizer design is simply omitting this staticizer [13]. If the designer does that, it is equivalent to using a staticizer whose resolution time is infinite, and the error rate of the system will increase by many orders of magnitude. Second, the node $sv\_$ and the staticizer can be replaced by a normal S-R latch; this will improve the resolution time compared to the weak-feedback staticizer. This design is shown schematically in Fig. 37. Finally, we should note that the designer who wishes to use domino logic between $sv\_$ and the next clock phase must proceed

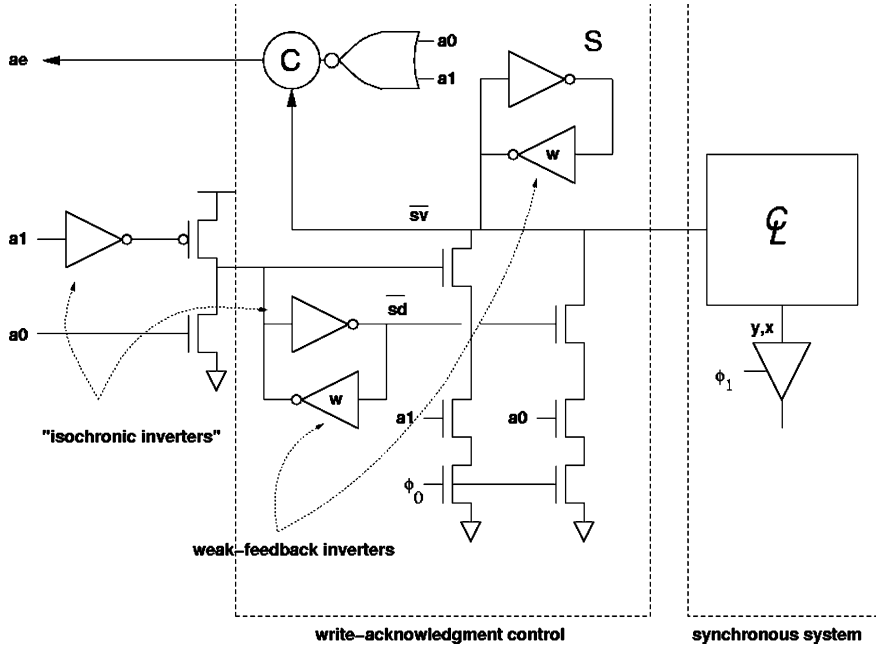

Fig. 35. *Detailed implementation of asynchronous-synchronous interface controller.*

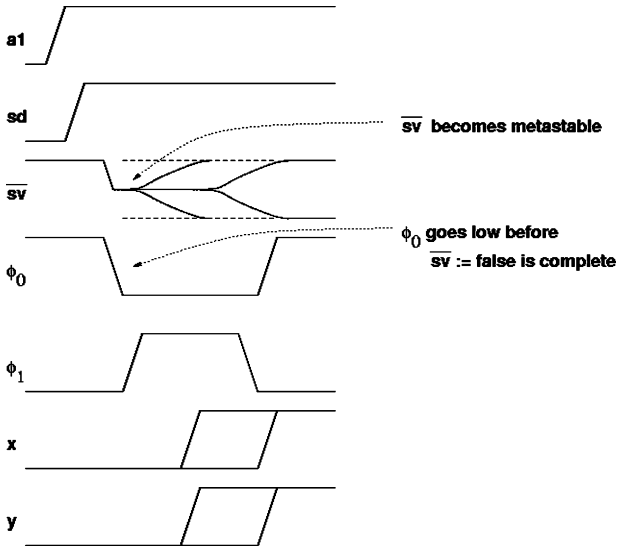

Fig. 36. *Timing diagram describing events leading up to synchronization failure.*

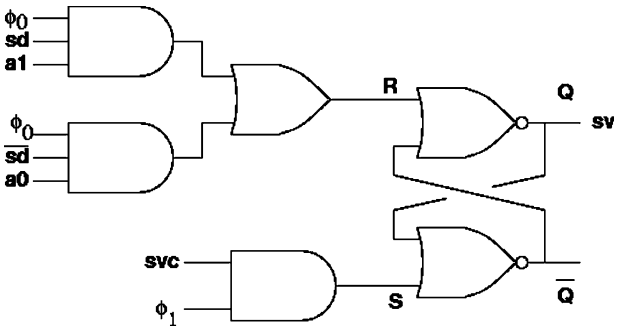

Fig. 37. *Use of S-R latch to improve metastability resolution time.*

very carefully, as in our current design *sv_* can have a downwards glitch.

The asynchronous-synchronous interface presented here obeys the Hippocratic oath of logic design: first, introduce no new timing races. The design has no necessary race conditions that are not inherent in the clocking methodology. There are three sources of timing races: races between different delays in the combinational logic; races due to the possibility of synchronization failure; and races within the clock generator. The first and third categories are inherent in the design style, the second is a necessary result of solving the synchronization problem, and the first and second can be handled by slowing down the clock, should they cause problems. This is the best that we can do. A corollary of the careful adherence to the logic designer's Hippocratic oath is that a synchronous system can be designed so that *any* timing variation in the system, including one resulting from metastability, can be absorbed by simply slowing down the clock.

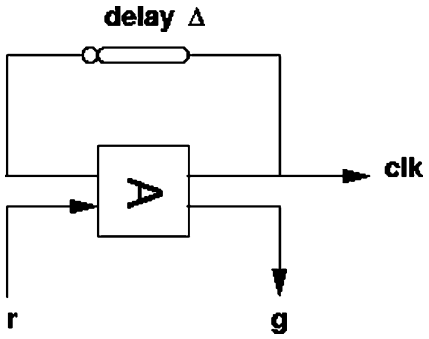

Fig. 38. *Safely stoppable clock oscillator.*

Finally, should the error probability of the circuit be too high for comfort at the lowest clock rate we are willing to tolerate, we can simply add additional latches where the $sv\_$ wire enters the synchronous domain; this will bring the probability of failure arbitrarily low, at the cost of adding synchronization latency. In as simple a circuit as this one, the extra latency also hurts the data throughput on the synchronized channel, but with more sophisticated techniques ([13], [47], [57] . . .), this can be avoided.

A. Stoppable-Clock GALS

The alternative to using synchronizers where the asynchronous signal enters the clock domain is to permit the asynchronous signal to stop the clock. In such a scheme, the asynchronous part of the system can treat all variables on the synchronous side as registers; as long as the writes to the variables are complete when the asynchronous side releases the clock, with the caveats mentioned below, there is no risk of misbehavior.

The idea behind stoppable, or pausable, clocks is to build an oscillator whose every cycle involves an arbitration against an external request r. Every time the clock wins the arbitration, it ticks; every time it loses, it stops and waits for the asynchronous request to be withdrawn. The circuit is shown schematically in Fig. 38; the grant g signals to the asynchronous part that the clock is stopped and the asynchronous part has exclusive access to shared variables (in this case, sv). Of course the arbiter itself introduces the possibility of metastability, but in this case the entire system simply waits for the metastability to be resolved; neither the clock nor the asynchronous part of the system is permitted to proceed until that happens. Normally the delay is implemented with a string of inverters, turning the stoppable-clock generator into a stoppable ring oscillator.

1) Stoppable-Clock Asynchronous-Synchronous Interface: To complete the stoppable-clock interface, we need to specify the clock oscillator, the clock generator, and the asynchronous state machine for the data transfers. A block diagram of the design is shown in Fig. 39. We keep the design as similar as possible to the synchronizer-based described in Section IX-A. We assume a clock generator of the type described by Mead and Conway [31]. We arrange it so that when the oscillator is in the stopped state (ϕ_x low), ϕ_0 is high and ϕ_1 is low: this choice matches the design of the synchronizer of Section IX-A. With this choice, the specification of the asynchronous controller becomes the following:

$$*[[\neg sv]; ae\uparrow; [a0 \longrightarrow sd\downarrow; r\uparrow \; []a1 \longrightarrow sd\uparrow; r\uparrow]; [g]; \\ sv\uparrow; ae\downarrow; [\neg a0 \wedge \neg a1]; r\downarrow; [\neg g]].$$

It is possible to increase the performance of this design slightly by postponing the wait for input neutrality, $[\neg a0 \wedge \neg a1]$ so that it overlaps with the reset phase of the arbiter, but the HSE we show here has a satisfyingly simple implementation, shown in Fig. 40; it is identical to the asynchronous controller for the synchronizer-based solution, except that the C-element driving ae has one additional input.

As is true of all the other asynchronous-synchronous interfaces described in this paper, the single-rail data encoding used on the synchronous side of the interface introduces timing assumptions in the asynchronous side of the interface. However, these timing assumptions can always be satisfied by slowing down the clock speed, exactly as in a synchronous system; in fact it is these timing assumptions that are the defining characteristic of synchronous design, as only delay-insensitive design styles are free of such timing assumptions. It is important to keep in mind the essential difference between this kind of timing "race" and the unavoidable one that results from the intrinsic nature of the requirement that data be synchronized with a free-running clock: the former type of timing race can be satisfied by manipulating physical parameters, usually in some obvious way; and the latter type of timing race cannot under any circumstances be avoided in its entirety as long as the clock-synchronization requirement is maintained. The type of circuit described in this section, as it does not attempt to synchronize data to a free-running clock, is completely free of the latter type of timing race.

2) Interfaces Using Edge-Triggered Methodology: In the previous section, we explored the design of asynchronous-synchronous interfaces using a clocking methodology with multiphase nonoverlapping clocks. Similar designs are possible if we are using edge-triggering clocks in our synchronous methodology, but there are some subtleties to bear in mind, resulting from the timing characteristics of edge-triggered flip-flops. Unlike the situation with nonoverlapping clocks, each edge-triggered memory element, called an edge-triggered flip-flop, itself has a timing race.

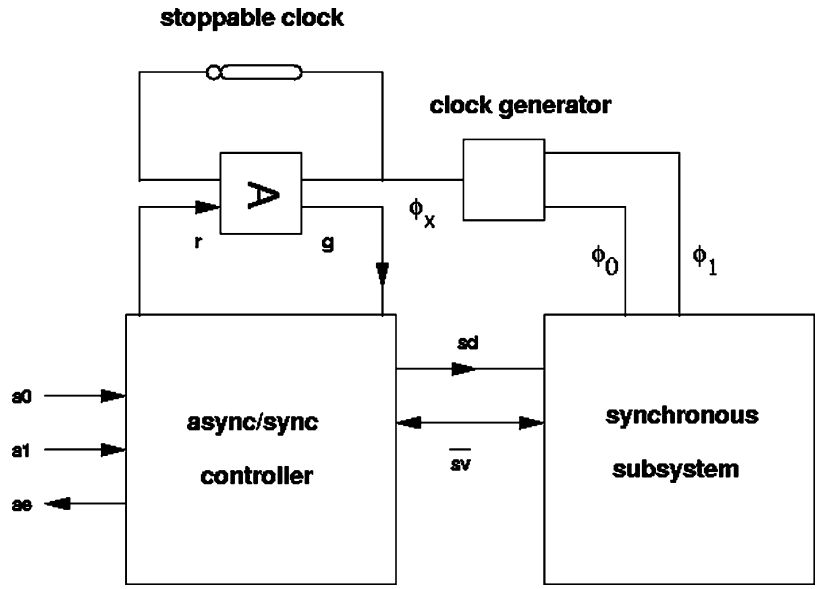

Fig. 39. *Block diagram of stoppable-clock interface for multiphase clocks.*

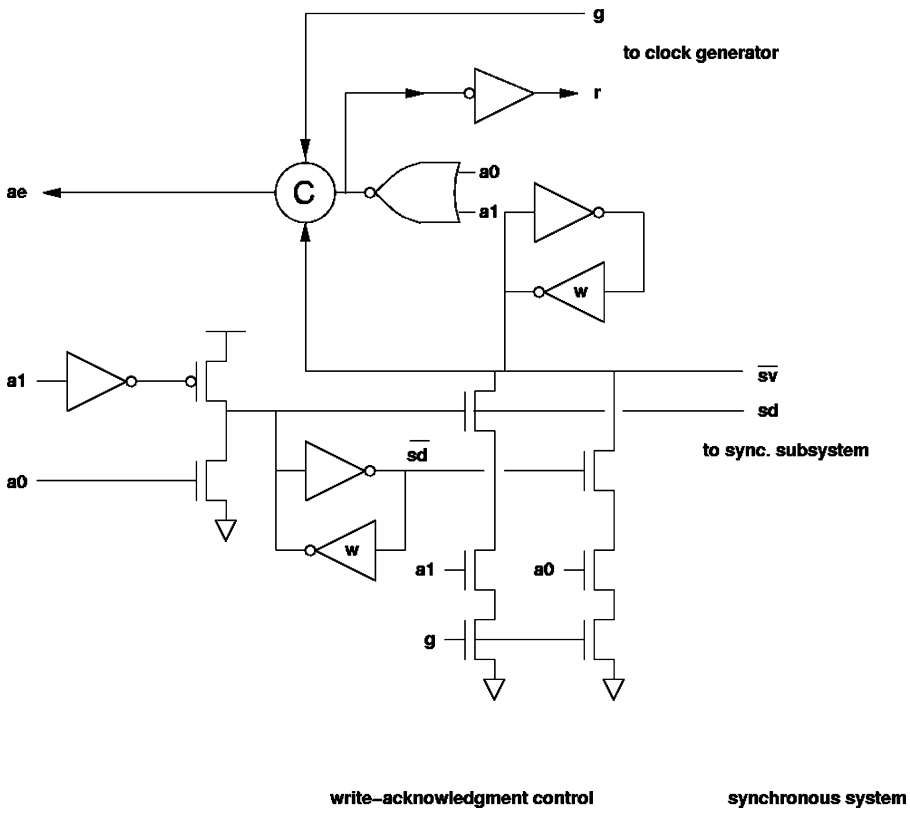

Fig. 40. *Circuit diagram of controller for stoppable-clock interface.*

The timing races in edge-triggered logic give rise to timing constraints called *setup* and *hold* times, whereas nonoverlapping logic only has the setup constraint.

Space constraints preclude us from discussing the details of the synchronization issues that arise when using edge-triggered logic, but the reader is urged to exercise

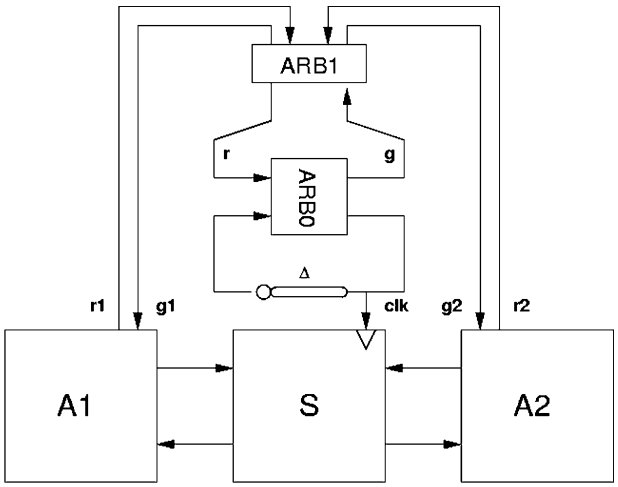

Fig. 41. *Sharing a stoppable clock from several interfaces; tree arbiter version.*

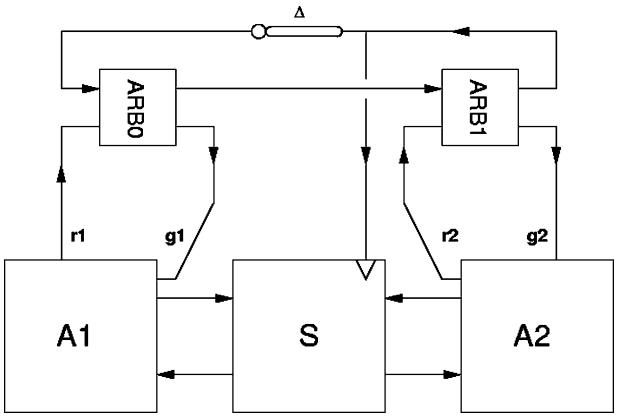

Fig. 42. *Sharing a stoppable clock from several interfaces; sequential arbitration version.*

care. The most important difference between edge-triggered logic and multiphase clocking is that there is a time near each active clock edge at which inputs are not permitted to change; contrast this with the situation in multiphase systems, where each variable is required to be quiescent only at the falling edge of the clock used to sample it and not at other clock edges. The simplest solution is to use the inactive clock edges to drive the asynchronous state machine and reserve the active clock edge for the synchronous logic. With this in mind, it is straightforward to derive correct synchronizers and stoppable clocks for the edge-triggered methodology.

B. Using Stoppable Clocks With Multiple Interfaces

It is rarely interesting to design a GALS system where modules have only one-way communication; it is as good as always true that if module P sends module Q messages, then it expects some sort of response (directly or indirectly).

Where multiple interfaces are involved, building a GALS system using synchronizers poses no new difficulties. If we are using stoppable clocks, however, we need to introduce a mechanism by which one or several asynchronous subsystems can stop the clock of the synchronous subsystem. To do this, we can combine the stoppable clock generator of Fig. 37 with the multiplexed arbiter presented in Section VIII-D, as shown in Fig. 41, where ARB1 is an instance of the multiplexed arbiter, whereas ARB0 is the normal arbiter used in the stoppable-clock generator.

From our earlier analysis of the multiplexed arbiter, we know that at most one of $g1$ and $g2$ can be active at any time; this arbiter sequences the accesses to the synchronous logic block by the asynchronous units $A1$ and $A2$. It is also true that the standard implementation of the arbiter ARB0 in the clock generator is fair; therefore, if, say, $A0$ initiates a data transfer while the master clock is high and does not complete it within half a clock, the clock will win the ARB0 arbitration as soon as the request r is withdrawn. Therefore, slow asynchronous accesses to the synchronous subsystem: 1) can occur only interspersed with synchronous cycles and 2) are mutually exclusive.

A different implementation with multiple interfaces is shown in Fig. 42. Here, the arbiters are arranged sequentially instead of in a tree; both ARB0 and ARB1 are standard arbiters. The result of this is that $A0$ and $A1$ can access S at the same time. The drawback of this approach is that while it is simpler, it results in extra delay in the clock generator; as long as desired clock speeds are low and the number of external interfaces on S is small, this delay can easily be compensated for by changing the clock delay Δ, but in other cases, it may become limiting. Accordingly, the tree arbiter is more suitable for systems with many channels or high speeds. It is possible, but difficult, to design a tree arbiter that can permit multiple simultaneous access. (This problem is an instance of "readers and writers mutual exclusion.") Finally, Muttersbach *et al.* describe an implementation of a shared-interface controller implemented with timing assumptions [35].

C. Zero Probability of Failure With Free-Running Clocks

While we have seen real advantages in the use of stoppable clocks in GALS systems, it must be admitted that most systems that can be called "GALS" do not actually use such clocks but instead use the more traditional free-running-clock-and-synchronizers approach. One of the reasons for this was discovered by researchers at ETH Zürich [58]: the control of delays by use of an off-chip clock offers an extremely versatile and convenient means for adjusting the clock speed; this is the case because of all physical parameters (voltage, current,

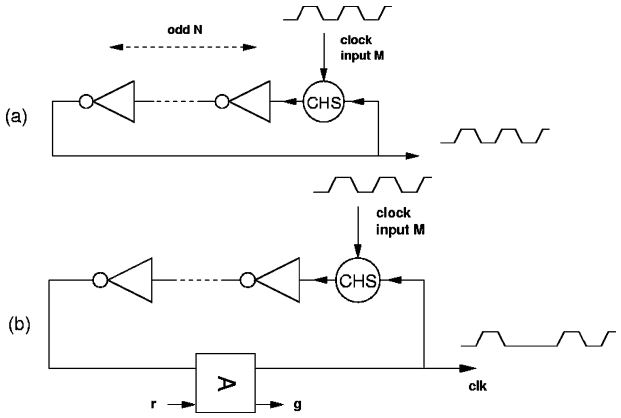

Fig. 43. *(a) Weakly synchronized "ring oscillator." (b) Weakly synchronized stoppable-clock generator.*

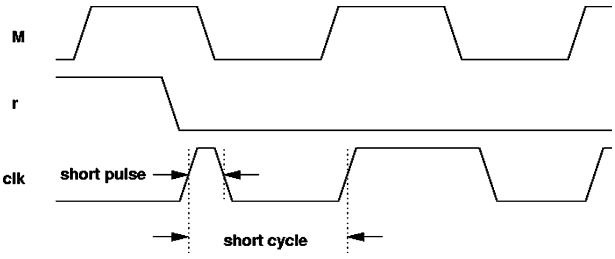

Fig. 44. *Timing diagram showing weakly synchronized stoppable-clock generator generating a very short clock pulse.*

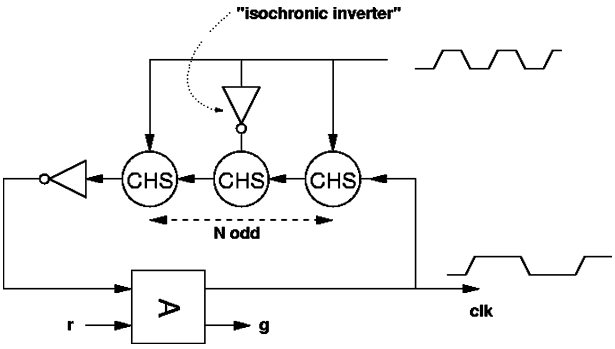

Fig. 45. *Improved weakly synchronized stoppable-clock generator.*

temperature, etc.) that can be adjusted, *time* is the one that most easily, conveniently, accurately, and cheaply can be adjusted over the largest range necessary in CMOS VLSI systems. In other words, the ring oscillators are difficult to control and must themselves be designed with utmost care if they are to be controllable over the entire range of speeds of interest in the CMOS VLSI systems; this issue is likely to become even more important if the systems are, say, used under dynamic voltage scaling (as might be appropriate for energy-efficient GALS systems).

It appears that what is desired is a stoppable clock that runs at a speed determined by an external clock. Of course this is not difficult to establish using, say, a phase-locked loop with a disable signal. However, the area, delay, and energy overheads involved in building such a device are very large, and furthermore, there are many pitfalls involved in the design of phase-locked loops, especially if they are to be stopped and started frequently. Is there anything simpler that can be done, always of course keeping in mind our Hippocratic oath of not adding any new timing races? While at first the outlook for such a circuit may seem bleak, the clock/handshake circuit of Section VIII-G suggests that there may be a way out. This circuit, we remember, can take a train of clock signals and turn it into a sequence of handshakes in such a way that there are never more handshakes than clock pulses. If we build a ring oscillator and put a clock/handshake circuit into it as shown in Fig. 43(a), we obtain a "ring oscillator" that runs at least no faster than the provided clock signal. Therefore, we can build a clock-controller ring oscillator simply by building a fast ring oscillator and slowing it down with a clock/handshake circuit. Of course, this means we can also make a stoppable clock out of it, as shown in Fig. 43(b). Unfortunately, this clock generator has a property that makes it unsuitable for solving our problem; namely, while it cannot produce more clock pulses than it is given, it can still produce *shorter* clock pulses, as we can see by studying the timing diagram in Fig. 44. A simple solution to the problem of the short clock pulses is to speed up the master clock M and use several clock/handshake circuits (Fig. 45). Effectively, the clock pulse of the ring oscillator "runs the gauntlet" between these clock/handshake circuits and cannot just happen to arrive late at more than one of them; the number of clock/handshake circuits should be odd, and the inverters from the clock are "isochronic inverters" that have to be fast compared to the other logic. With N clock/handshake circuits in sequence, we can lose at most $1/N$ of the high clock pulse, or $1/(2N)$ of the entire clock cycle. Therefore, we must run the system at $(N+1)/N$ of the speed it can work at without the stoppable clock, or at $(N+1/2)/N$ of that speed if we care only about the total length of the clock cycle (e.g., if we are using edge-triggered flip-flops in the synchronous part).

D. A Real-World Effect: Clock-Tree Delays

Another reason that stoppable clocks are little used in practice is that designers often desire to drive very large synchronous blocks with the clocks generated by the controllers described in this paper. Such large blocks may have large internal clock loads, necessitating that the clocks be driven by clock trees. In such a situation it is important to realize that a flip-flop with "zero hold time"

relative to its own clock has a positive hold time relative to the clock made by the clock generator, and many of the techniques we have studied will not work without delay matching on the other signals (e.g., the request and grant lines to the arbiter). A simpler solution, which has not been explored much, is to make the synchronous islands smaller, using the kinds of simple circuits described in this paper, so that they do not require clock trees, and so that the techniques can be applied exactly as described.

E. Using Synchronizers in GALS Systems

Most practical GALS systems use a free-running master clock with synchronizer interfaces, as this is usually the most convenient approach, especially if the designer is uncomfortable with designing clock-generation circuitry. As we have seen, designing the necessary synchronizer interfaces is a nontrivial task with several subtle challenges; also, the digital designer is not well prepared to meet these challenges, as his experience generally is carefully tailored to exclude all the situations where "digital" signals are switching between the setup- and hold-constraint times.

Unfortunately, the problems do not even end with the subtleties of getting the local synchronizer circuits right. Other problems can occur that can only be analyzed globally for a system. An example of this, first mentioned by Stucki and Cox, is that the synchronization error rates derived for the synchronizing latches (or flip-flops) are based on simple statistics assuming signals whose edge timings are completely uncorrelated. If the system we are designing is an SoC with multiple clock domains all driven off a single master (but with unknown interdomain skew), the assumption that the signal timings are uncorrelated no longer holds; the same conclusion is reached if we consider a system with multiple independent on-chip oscillators. For this reason, Stucki and Cox suggest that SoC designers would do well to consider stoppable-clock schemes, which of course have no synchronization failures, more often. All we can do is concur.

Other abuses of local analysis have appeared in published designs. For instance, there have been designs where different subsystems are connected to each other via FIFOs, and where the probability of synchronization failure can be guaranteed to be zero as long as the FIFOs are neither empty nor full [57]. While this is certainly true, and while it is usually the case that the designs are correct even when the FIFOs are drained or filled (in the sense that they do not fail more often than necessary), one can question the usefulness of the property that the FIFOs are known not to fail while they are half full. The problem is: how can the system be designed to keep the FIFOs from getting empty or full? The answer is, unfortunately, that it cannot. Either the clocks ϕ_x and ϕ_y are completely synchronized (in which case the system is not really a GALS system, but a special case of a synchronous system that will either always work or always fail, depending on the timing characteristics of the modules—see above), or else one (say x) will run slightly faster than the other (y). Therefore, module x will consume its inputs faster than y can send them or generate its outputs faster than y can consume them, or both. Eventually the FIFO carrying data from y to x must empty, or the FIFO carrying data from x to y must fill completely.

The preceding discussion is an instance of the general principle that synchronizations can be reused by exploiting temporal coherency, and many synchronization schemes are based on this principle. We can divide such synchronization strategies into two parts: the synchronization event, and the data transfer. A prepared data transfer block needs only a single synchronization to pass from one clock domain (or from the asynchronous domain) into another clock domain, which limits the *latency* of the transfer to be greater than some minimum value, given by our tolerance for synchronization failure. If the data is properly arranged beforehand, this single synchronization event can be used to permit an arbitrarily large amount of data to be transferred; therefore, the synchronizer does not constrain the *throughput* of the transfer; Stucki and Cox [47] described a design that takes advantage of this effect, using interleaved synchronizers.

The purpose of this paper was to expose the SoC architect to a comprehensive set of standard asynchronous techniques and building blocks.

X. CONCLUSION

The purpose of this paper was to expose the SoC architect to a comprehensive set of standard asynchronous techniques and building blocks for SoC interconnects and on-chip communication. Although the field of asynchronous VLSI is still in development, the techniques and solutions presented here have been extensively studied, scrutinized, and tested in the field—several microprocessors and communication networks have been successfully designed and fabricated. The techniques are here to stay. The basic building blocks for sequencing, storage, and function evaluation are universal and should be thoroughly understood. At the pipeline level, we have presented two different approaches: one with a strict separation of control and datapath, and an integrated one for high throughput. Different versions of both approaches are used. At the system level, issues of slack, choices of handshakes and reshuffling affect the system performance

in a profound way. We have tried to make the designer aware of their importance. At the more fundamental level, issues of stability, isochronic forks, validity, and neutrality tests, state encoding must be understood in order for the designer to avoid the recurrence of hazard malfunctions that have plagued early attempts at asynchrony.

Many styles of asynchronous designs have been proposed. They differ by the type and extent of the timing assumptions they make. Rather than presenting and comparing them all in a necessarily shallow way, we have chosen to present the most "dasynchronous" approach—QDI—in some depth. While we realize very well that the engineer of an SoC has the freedom and duty to make all timing assumptions necessary to get the job done correctly, we also believe that, from a didactic point of view, starting with the design style from which all others can be derived is the most effective way of teaching this still vastly misunderstood but beautiful VLSI design method. ■

Acknowledgment

The authors would like to thank J. Dama, W. Jang, M. Josephs, J. L. Martin, M. Naderi, K. Papadantonakis, and P. Prakash for their excellent comments on the paper.

REFERENCES

[1] The "asynchronous" bibliography. (2004). [Online]. Available: http://www.win.tue.nl/async-bib/

[2] H. C. Brearley, "ILLIAC II: A short description and annotated bibliography," *IEEE Trans. Comput.*, vol. C-14, no. 6, pp. 399–403, Jun. 1965.

[3] C. H. K. v. Berkel and R. Saejis, "Compilation of communicating processes into delay-insensitive circuits," *Proc. Int. Conf. Computer Design (ICCD)*, 1988, pp. 157–162.

[4] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," *Dig. Tech. Papers 1989 IEEE Int. Conf. Computer-Aided Design*, pp. 262–265.

[5] S. M. Bums and A. J. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," in *Advanced Research in VLSI*, J. Allen and F. Leighton, Eds. Cambridge, MA: MIT Press, 1988.

[6] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. Comput.*, vol. C-22, no. 4, pp. 421–422, Apr. 1973.

[7] D. M. Chapiro, *Globally-Asynchronous, Locally-Synchronous Systems*, Ph.D. dissertation, Stanford Univ., Stanford, CA, 1984, Stanford CS Tech. Rep. STAN-CS-84-1026.

[8] T.-A. Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," in *Proc. Int. Conf. Computer Design (ICCD)*, 1987, pp. 220–223.

[9] F.-C. Cheng, "Practical design and performance evaluation of completion detection circuits," in *IEEE Int. Conf. Computer Design (ICCD)*, 1998, pp. 354–359.

[10] W. A. Clark, "Macromodular computer systems," in *AFIPS Conf. Proc. 1967 Spring Joint Computer Conf.*, 1967, pp. 335–336.

[11] J. Cortadella *et al.*, *Logic Synthesis of Asynchronous Controllers and Interfaces*. New York: Springer-Verlag, 2002.

[12] ——, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inf. Syst.*, vol. E80-D, pp. 315–325, 1997.

[13] W. J. Dally and J. W. Poulton, *Digital Systems Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[14] S. B. Furber *et al.*, "A micropipelined ARM," in *Proc. VII Banff Workshop: Asynchronous Hardware Design*, 1993.

[15] ——, "AMULET2e: An asynchronous embedded controller," in *Proc. Async '97*, 1997, pp. 290–299.

[16] J. D. Garside, "Processors," in *Principles of Asynchronous Circuit Design: A Systems Perspective*, J. Sparse and S. Furber, Eds. Boston, MA: Kluwer, 2001, ch. 15.

[17] H. v. Gageldonk *et al.*, "An asynchronous low-power 80c51 microcntroller," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 96–107.

[18] M. R. Greenstreet, "Real-time merging," in *Proc. Fifth Int. Symp. Advanced Research in Asynchronous Circuits and Systems: ASYNC99, Barcelona, Spain, 19–21 April 1999*, Los Alamitos, CA: IEEE CS Press, 1999.

[19] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, 1978.

[20] D. A. Huffman, "The synthesis of sequential switching circuits," in *Sequential Machines: Selected Papers*, E. F. Moore, Ed. Reading, MA: Addison-Wesey, 1964.

[21] C. Kelly, IV, V. Ekanyake, and R. Manohar, "SNAP: A sensor network asynchronous processor," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 2003, pp. 24–35.

[22] A. M. Lines, "Pipelined asynchronous circuits," M.S. thesis, California Inst. Technol., Pasadena, 1997.

[23] R. Manohar and A. J. Martin, "Quasi-delay-insensitive circuits are turing-complete," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, Fukushima, Japan, 1996.

[24] A. J. Martin, "The design of a self-timed circuit for distributed mutual exclusion," *Proc. 1985 Chapel Hill Conf. VLSI*, H. Fuchs, Ed., pp. 245–260.

[25] ——, "Programming in VLSI: From communicating processes to self-timed VLSI circuits," in *Concurrent Programming (1987 UT Year of Programming Institute on Concurrent Programming)*, C. A. R. Hoare, Ed. Reading, MA.: Addison-Wesley, 1989.

[26] ——, "The limitations to delay-insensitivity in asynchronous circuits," *Proc. 6th MIT Conf. Advanced Research in VLSI*, W. J. Dally, Ed., 1990, pp. 263–278.

[27] A. J. Martin *et al.*, "The design of an asynchronous microprocessor," *Proc. Decennial Caltech Conf. Advanced Research in VLSI*, C. L. Seitz, Ed., 1991, pp. 351–373.

[28] ——, "The Design of an Asynchronous MIPS R3000 Processor," in *Proc. 17th Conf. Advanced Research in VLSI*, Los Alamitos, CA: IEEE CS Press, 1997.

[29] A. J. Martin, "Synthesis method for self-timed VLSI circuits," in *Proc. 1987 IEEE Int. Conf. Computer Design (ICCD 87)*, pp. 224–229.

[30] A. J. Martin, M. Nyström, and C. G. Wong, "Three generations of asynchronous microprocessors," *IEEE Des. Test Comput. (Special Issue on Clockless VLSI Design)*, vol. 20, no. 6, pp. 9–17, Nov./Dec. 2003.

[31] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[32] R. E. Miller, *Switching Theory*. New York: Wiley, 1965, vol. 2, ch. 10.

[33] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," *Proc. Int. Symp. Theory of Switching*, 1959, pp. 204–243.

[34] C. J. Myers, *Asynchronous Circuit Design*. New York: Wiley, 2001.

[35] J. Muttersbach, T. Villiger, and W. Fichner, "Practical design of globally-asynchronous locally-synchronous systems," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 2000, pp. 52–59.

[36] T. Nanya *et al.*, "TITAC: Design of a quasi-delay-insensitive microprocessor," *IEEE Des. Test Comput.*, vol. 11, no. 2, pp. 50–63, Summer, 1994, IEEE.

[37] ——, "TITAC-2: A 32-bit scalable-delay-insensitive microprocessor," in *Proc. HOT Chips IX*, 1997, pp. 19–32.

[38] S. M. Nowick, M. B. Josephs, and C. H. van Berkel, "Special issue: Asynchronous circuits and systems," *Proc. IEEE*, vol. 87, no. 2, pp. 217–396, 1999.

[39] M. Nyström and A. J. Martin, "Crossing the synchronous-asynchronous divide," presented at the *Workshop Complexity Effective Design*, Anchorage, AK, 2002.

[40] ——, *Asynchronous Pulse Logic*. Boston, MA: Kluwer, 2001.

[41] P. Prakash and A. J. Martin, "Slack matching quasi-delay-insensitive circuits," in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 2006, pp. 195–204.

[42] M. Renaudin, P. Vivet, and F. Robin, "ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 22–31.

[43] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P. Fang, "Q-modules: internally clocked delay-insensitive modules," *IEEE Trans. Comput.*, vol. 37, no. 9, pp. 1005–1018, Sep. 1988.

[44] S. Rotem *et al.*, "RAPPID: An asynchronous instruction length decoder," *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 60–70.

[45] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980, ch. 7.

[46] J. Sparsø and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Boston, MA: Kluwer, 2001.

[47] M. J. Stucki and J. J. Cox, "Synchronization strategies," in *Proc. 1st Caltech Conf. Very Large Scale Integration*, C. L. Seitz, Ed., 1979, pp. 375–393.

[48] I. E. Sutherland and S. Fairbanks, "GasP: A minimal FIFO control," in *Proc. 10th Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 2004, pp. 46–53.

[49] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160 ns 54 b CMOS divider," *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, Nov. 1991.

[50] *PDP-11 Peripherals and Interfacing Handbook*, Digital Equipment Corp., Maynard, MA, 1971, DEC part no. 112-01071-1854 D-09-25.

[51] *PDP-11 Bus Handbook*, Digital Equipment Corp., Maynard, MA, 1979, DEC part no. EB-17525-20/79 070-14-55.

[52] *Arithmetic Processor 166 Instruction Manual*, Digital Equipment Corp., Maynard, MA, 1960.

[53] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.

[54] Small computer system interface—2, Tech. Comm. T10, Amer. Nat. Stand. Inst., Draft report X3T9.2/86-109, 1989.

[55] C. G. Wong and A. J. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proc. ACM/IEEE Design Automation Conf.*, 2003, pp. 508–513.

[56] L. A. Glasser and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*. Reading, MA: Addison-Wesley, 1985.

[57] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 8, pp. 857–873, Aug. 2004.

[58] F. K. Gürkaynak *et al.*, "GALS at ETH Zurich: Success or failure?" in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 2006, pp. 150–159.

ABOUT THE AUTHORS

Alain J. Martin (Member, IEEE) graduated from the Institut National Polytechnique de Grenoble, France.

He is a Professor of Computer Science at the California Institute of Technology (Caltech), Pasadena. His research interests include concurrent computing, asynchronous VLSI design, formal methods for the design of hardware and software, and computer architecture. In 1988, he designed the world-first asynchronous microprocessor.

Prof. Martin is a member of the Association for Computing Machinery.

Mika Nyström (Member, IEEE) received the S.B. degree in physics and electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1994 and the M.S. and Ph.D. degrees in computer science from the California Institute of Technology (Caltech), Pasadena, in 1997 and 2001, respectively.

He is a Research Scientist in the Department of Computer Science at Caltech. His graduate research covered a wide range of topics relevant to asynchronous and other high-speed VLSI systems; his dissertation focused on design issues encountered in a novel family of ultrafast asynchronous circuits.