Asynchronous Techniques for VLSI System Design

Alain J. Martin and Mika Nyström
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA
A short version of this paper is to appear in *Proceedings of the IEEE*
Special Issue on Systems on Chip

February 16, 2006

Introduction

It is now generally agreed that the large VLSI systems (*systems-on-chip*) 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 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, this paper will place 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 first section gives a few brief history and the main definitions of the different asynchronous logics according to their timing assumptions. Section 2 introduces the computational models and languages used in this paper to describe and construct asynchronous circuits. Section 3 is a brief theory of quasi-delay-insensitive logic. It may be skipped at first reading. Section 4 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 5. Section 6 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 7 describes several implementations of buses. Section 8 briefly explains how a system component can be decomposed into a fine-grain collection of asynchronous modules. Section 9 deals with issues of arbitration, and synchronization. Section 10 presents the asynchronous/synchronous interfaces needed in a GALS system.

1 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[59]. The "macromodule" experiment at Washington University in St. Louis 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 work appeared around the same period, in particular the works of D.A. Huffman[22], D.E. Muller[37], and also S.E. Unger's work published in book form in 1969[60]. The approach presented in this paper for asynchronous logic synthesis can be viewed as an extension of Muller's work. The interested reader may start with Chapters 5 and 6 in Chris J. Myers's book[38]. An excellent but older presentation of Muller's work can also be found in [36].

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[53].

C.L. Seitz's chapter on "System Timing" in Carver Mead and Lynn Conway's epoch-making 1980 "Introduction to VLSI Systems" revived the research community's interest in the topic—if not the industry's interest[50, 35]. Most of the research that would follow in the 1980's would build on the observation made in Mead and Conway that a VLSI chip is a fine-grain concurrent computation. This new view of the field, incorporating principles and programming models from concurrent computing (in particular message-passing distributed computing), was a break from previous approaches radical enough to be considered a fresh start.

The *metastability* phenomenon, inherent in the arbitration between two completely unsynchronized signals, has been the cause of some of the most interesting intellectual quarrels in the field. Many practitioners of the day simply refused to admit that the problem was unsolvable (i.e., metastability cannot be avoided), by the same token refusing to recognize that some of their clocked architectures were plagued with timing glitches caused by metastability. The pioneering work of Charlie Molnar and his colleagues at Washington University was instrumental in explaining metastability[6].

The first "modern" synthesis methods appear around the mid-1980 with the Caltech program-transformation approach[28] 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[5, 3, 4]. *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[31]. It was followed by the first "Amulet" (a family of asynchronous clones of the ARM processor) from the University of Manchester in 1993[16], the TITAC, and 8-bit microprocessor from the Tokyo Institute of Technology in 1994[40], and the Amulet2e and TITAC-2 in 1997[17, 41]—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[32]. 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[34]. Marc Renaudin and his group at Grenoble ported the Caltech MiniMIPS building blocks to standard cells to use in a 16-bit RISC[47]. Other asynchronous chip experiments include the design of a fast divider at Stanford in 1991 by Ted Williams[56], and an instruction-length decoder for the Pentium by a research group at Intel in 1999[49].

Recent asynchronous-processor projects have emphasized the low-power advantage of asynchronous logic: Low-power asynchronous 8051 microcontrollers have been designed at Philips[19] and Caltech[34]. Rajit Manohar and his group at Cornell have been designing low-power processors (the SNAP and BitSNAP) for sensor network applications[24, 15].

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[39].

Several books on asynchronous logic have been published. Among them, our favorites are [38], [11], and [52]. A special issue of the *Proceedings of the IEEE* also gives a good overview of the state of the

art[42]. The book by Dally and Poulton, although not specifically about asynchronous systems contains an excellent chapter on synchronization[13]. The on-line *Asynchronous Bibliography* gives an almost complete bibliography of the field todate[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 (or timing information) to implement sequencing. Since a main advantage of asynchronous circuits is their insensitivity to delay variations, it seems self-defeating to first remove the clock and then reintroduce delay requirements. Therefore, the "baseline" against which other methods should be compared is the one that introduces the fewest timing assumptions.

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 [50]. It would be formalized soon after in the theory of "Trace Theory and VLSI design" developed at the University of Eindhoven by Martin Rem, Jan van de Snepscheut, J. T. Udding, and Jo Ebergen [46, 51, 55, 14]. For a while, it was believed that it should be possible to design entirely delay-insensitive circuits, and this belief is reflected in the number of occurrences of the term in articles' titles of the eighties. Unfortunately, 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[30]. 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[30]. Those forks are called *isochronic*. We will return to the definition of isochronic fork when we introduce the model in more detail.

Asynchronous circuits with the only delay assumption of isochronic forks are called quasi delay-insensitive or 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[37].

Similarly, *self-timed* circuits are asynchronous circuits in which all forks that fit inside a chosen physical area called *equipotential region* are isochronic[50].

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 [38]) 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* ([44]) and GasP ([54]).

2 SoCs as Distributed Systems

Digital systems, in particular systems-on-chip, 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. Synchronous 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 CHP (*Communicating Hardware Processes*). It is based on C.A.R. Hoare's first version of CSP([21]), 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.

2.1 Computational Models

Programming languages are the best formalisms to describe and design large computational systems like SoCs. Other methods, in particular graphical representations like Petri Nets and state-transition graphs (STG), have the intuitive advantages of pictures but do not scale to large systems, and do not express integer arithmetics and boolean logic easily. 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. Starting with a coarse-grain concurrent system, or even with a sequential CHP description, a design is first refined at the CHP level to increase concurrency (e.g., through pipelining). This first step produces a concurrent system of CHP processes. Each process is then refined through a second stage of transformations into a representation that manipulates boolean variables only. The notation used for this representation is called HSE; it has the control structure of CHP but integers are replaced with collections of booleans and channels are replaced with handshake protocols on boolean variables. Finally, a third stage of transformations leads to a network of gates or CMOS transistors. The notation for this level of representation is called PRS. The PRS notation does not have a notion of sequential execution, since only concurrency is a primitive operation in hardware. Hardware description languages and modeling formalisms are not the subject of this paper. The languages and notations used in this paper, although slightly unusual, should not be a hurdle to the reader.

2.2 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.

2.2.1 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 *static 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. The *probe* on port C, denoted $\overline{C}$, is a boolean primitive that tests whether a communication action on the other port of the channel is pending. For example, assume $p1$ reaches communication R and is blocked at R waiting for $p2$ to reach L. Then $p2$ can use the probe $\overline{L}$ to detect that $p1$ has reached R.

2.2.2 Variables

A variable is either a boolean, or a (finite-range) integer, or a symbol. Variables can be grouped into arrays or records. An integer variable is also automatically declared as an array of booleans. 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 the assignments $b := true$ and $b := false$, respectively. In HSE and PRS, all variables are boolean.

2.2.3 Sequential and Parallel Compositions

CHP and HSE provide two composition operators: the sequential operator $S1; S2$ and the parallel operator. Unrestricted use of the parallel composition operator 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. 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. For reason of efficiency, CHP also allows a limited form of parallel composition *inside* a process, denoted by the comma, as in $S1, S2$. The comma is restricted to program parts that are *non-interfering*: In the parallel composition $S1, S2$, if $S1$ changes the value of variable x (writes x), then $S2$ does not use x (neither reads x nor writes x).

2.2.4 Selection, Wait, and Repetition

The *selection command* $[B1 \to S1 [] B2 \to S2 [] \ldots]$ is a generalization of the if-statement. It has an arbitrary number (at least one) of clauses, called "guarded commands", $Bi \to Si$ 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 $[B1 \to S1 \mid B2 \to S2]$.

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 $[B \to skip]$, 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 non-terminating repetition $*[S]$ that repeats S forever.

2.2.5 Pipeline Slack and Slack Matching

Slack matching is an optimization by which simple buffers are inserted into a system of distributed processes to increase the throughput. The static slack of a pipeline is the maximum number of messages the pipeline can hold. A pipeline 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 [25, 45].) 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. Because of the importance of slack matching on the performance optimization of a system, several alternative implementations of a given component will be compared in terms of their effect on the slack of the component. (See [25, 45].)

2.3 Modeling System Components: HSE

At the level of each component (process), the CHP design 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 a HSE program is as follows:

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

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 ..."

2.4 Modeling Circuits: Production Rules

The computational model used at the circuit level is called PRS for *production-rule set*. 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 output of a gate is connected to the low voltage level (the ground, used to represent the boolean value false) by a transistor network (the pull-down network) and to the high voltage level (Vdd, representing the boolean value true) by another transistor network (the pull-up network). The two networks together form the gate as shown in Figure 1. A transistor is modeled as a switch controlled by the boolean value of the gate of the transistor.

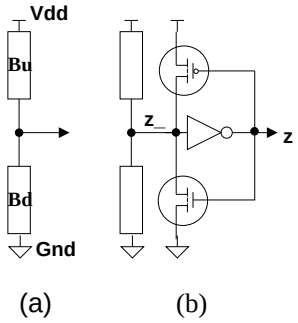

Figure 1: The pull-up and pull-down networks implementing a CMOS logic gate: (a) combinational gate, (b) state-holding gate with a standard "staticizer" ("keeper"). The circled transistors are weak.

A pull-up/pull-down network is a network of switches that connects or disconnects the output node of the operator and either the Vdd node or the ground node. There is a well-known one-to-one correspondence between a switching network and a boolean expression in terms of the variables controlling the switches. Let Bu be the boolean expression representing the pull-up network and Bd be the boolean expression representing the pull-down network, and let x be the output of the operator. If Bu holds, a conducting path exists between Vdd and x; if the path remains conducting long enough, eventually the voltage of x is pulled up to Vdd, i.e., x is assigned the value true. And similarly when Bd holds, x is assigned the value false. Those behaviors are formalized by the two *production rules*:

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

Definition 1 *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.*

Definition 2 *A production rule set (PRS) is the parallel composition of all production rules in the set.*

2.5 Boolean Operators

Once the PR set of a system has been synthesized, the complementary PRs, —the production rules that set and reset the same variable—are identified and implemented with (standard or non-standard) operators.

In this section, we introduce the small set of boolean operators used in the paper (except for the arbiter and synchronizer that will be introduced later). CMOS is the target technology, which imposes two main restrictions: (1) the number of transistors in series allowed in the implementation of a single gate is severely bounded; (2) in order to guarantee good signal transitions (slew rates), we use almost exclusively restoring logic, i.e., we avoid pass-transistors. Furthermore, we use p-transistors exclusively in pull-up chains and n-transistors exclusively in pull-down chains. With those restrictions, CMOS is an *inverting logic* .

For the sake of simplicity, we do not always describe the circuits in an inverting-logic form. When needed, we replace a variable x with its inverted version $x\_$.

Complementary PRs must be *non-interfering*, i.e., guards Bu and Bd cannot be true at the same time. (We return to this issue in the next section.) 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 implementing the PRs is *combinational* with the simple CMOS implementation of Figure 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*.

2.5.1 Combinational Gates

With the exception of the "write-acknowledge" we use standard combinational gates in this paper: inverter, nand, nor. Wire and forks are also described as operators, but since no transistor is used in their implementation, the amplification effect of the transistor is not used: they are not restoring gates. A *nand*-gate with inputs x and y and output z (denoted $z = (x \underline{nand}\ y)$) implements the PRs

$$\begin{aligned} x \wedge y &\rightarrow z\downarrow \\ \neg x \vee \neg y &\rightarrow z\uparrow . \end{aligned}$$

A *nor*-gate with inputs x and y and output z (denoted $z = (x \underline{nor}\ y)$) implements the PRs

$$\begin{aligned} x \vee y &\rightarrow z\downarrow \\ \neg x \wedge \neg y &\rightarrow z\uparrow . \end{aligned}$$

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. 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.

2.5.2 State-Holding Elements

In QDI design, dynamic implementations of state-holding elements are excluded because they rely on a timing assumption: the duration of a floating state is assumed short enough to guarantee that the values of the output variable does not change. Instead, state-holding gates are implemented in such a way that the output node is connected to the Vdd or the ground even in the states where neither Bu nor Bd holds. To do so, we have to add pull-up or pull-down conditions. For instance, to keep the true value of the output node z, we add pull-up Su, which amounts to adding PR $Su \rightarrow z\uparrow$. Since the PR is added only to maintain the value of z, its firing is vacuous; it does not change the value of z.

Adding vacuous PRs is also used to simplify circuits, in particular to transform state-holding gates into combinational ones. Such a transformation is always possible but is used only when the resulting combinational gate is simple. Otherwise, a standard "staticizing" circuit is added to the state-holding gate. The general idea is to have both the output zt and its inverse zf available, either by introducing an explicit inverter or by computing them separately. Then, $\neg zf \rightarrow zt\uparrow$ is added to keep zt high, $zf \rightarrow zt\downarrow$ is added keep zt low . Those PRs are obviously vacuous.

A difficulty is that when a transition fires to change the value of the output, there is an interference, a "fight", with the vacuous transition added to maintain the value. The transistor of the vacuous PR have to be implemented as "weak", i.e., with high enough resistance that the firing of the effective transition completes and changes the value of the output. A standard "keeper" or "staticizer" built in this manner is shown in Figure 1(b).

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

$$\begin{aligned} x \wedge y &\rightarrow z\uparrow \\ \neg x \wedge \neg y &\rightarrow z\downarrow . \end{aligned}$$

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 3-input C-element $z = x\ \underline{C}\ y\ \underline{C}\ t$ implements the PRs

$$\begin{aligned} x \wedge y \wedge t &\rightarrow z\uparrow \\ \neg x \wedge \neg y \wedge \neg t &\rightarrow z\downarrow . \end{aligned}$$

The *set-reset gate* with inputs s and r and output z has the PRs:

$$\begin{aligned} s &\rightarrow z\uparrow \\ r &\rightarrow z\downarrow \end{aligned}$$

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 $zt = \neg zf$, as follows:

$$\begin{aligned} s &\rightarrow zf\downarrow \\ \neg zf &\rightarrow zt\uparrow \\ r &\rightarrow zt\downarrow \\ \neg zt &\rightarrow zf\uparrow \end{aligned}$$

Adding the standard vacuous rules lead to two possible implementations. One with cross-coupled inverters implements the PRs

$$\begin{aligned} s \vee zt &\rightarrow zf\downarrow \\ \neg zf &\rightarrow zt\uparrow \\ r \vee zf &\rightarrow zt\downarrow \\ \neg zt &\rightarrow zf\uparrow . \end{aligned}$$

The other one, with cross-coupled nor-gates implements the PRs

$$\begin{aligned} s \vee zt &\rightarrow zf\downarrow \\ \neg zf \wedge \neg r &\rightarrow zt\uparrow \\ r \vee zf &\rightarrow zt\downarrow \\ \neg zt \wedge \neg s &\rightarrow zf\uparrow . \end{aligned}$$

The two extra transistors make the gates combinational and avoid the need for weak transistors.(See Figure 2.) We also use a non-standard operator, the *precharge function* (PCF). An example of a precharge function with inputs en, $x0$, $x1$, $y0$, $y1$ and output z is described by the following two PR pairs:

$$\begin{aligned} en \wedge (x0 \wedge y0 \vee x1 \wedge y1) &\rightarrow z1\downarrow \\ \neg en &\rightarrow z1\uparrow \\ en \wedge (x0 \wedge y1 \vee x1 \wedge y0) &\rightarrow z0\downarrow \\ \neg en &\rightarrow z0\uparrow \end{aligned}$$

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.

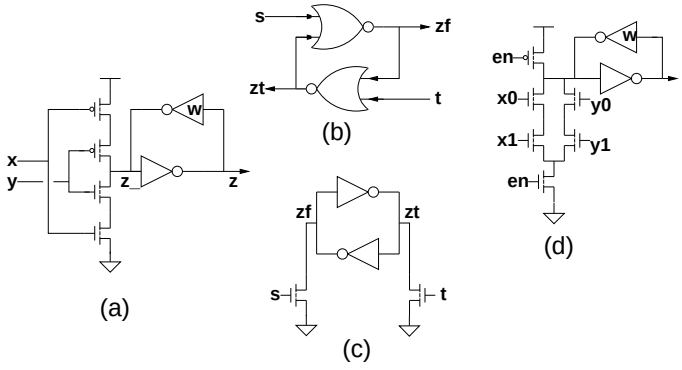

Figure 2: State-holding elements: (a) The C-element, (b) Nor-gate implementation of set-reset, (c) inverter implementation of set-reset, (d) An example of a precharge function. The cross-coupled inverters in (a) and (b) are standards "staticizers." The letter w indicates that the transistors of the inverter are weak.

3 A Brief Theory of QDI Logic

This section may be skipped at first reading beyond the definitions of stability and non-interference. However, the section is important as it explains why and under which conditions this method of asynchronous design produces circuits that are operating correctly, i.e., without *logic hazards*. A hazard is the possibility of an incomplete transition (a "glitch").[1]

3.1 Execution Model: Stability and Non-interference

Definition 3 *(1) An execution of production rule $G \rightarrow t$ is an unbounded sequence of firings. A firing of $G \rightarrow t$ when G is true amounts to the execution of t. A firing when G is false is a skip. The firing of a PR that changes the state of the computation is said to be effective; otherwise, the firing is said to be vacuous. (2) An execution of a production-rule set is the concurrent execution of all production rules in the set.*

A *transition* is an effective firing of a production rule. We use the predicate $R(t)$ to denote the result of transition t: $R(x\uparrow) \equiv x$, $R(x\downarrow) \equiv \neg x$.

The concurrent execution model is as general as possible: (1) transitions are not instantaneous: bringing a node from the low voltage level to the high voltage level or vice versa does take time; and (2) there is no notion of atomic action: the concurrent execution of *complementary transitions* $x\uparrow$ and $x\downarrow$ is undefined.[2]

Within such a model, 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) Guard G of $G \rightarrow t$ may cease to hold before transition t has completed, as the result of a concurrent transition invalidating G. (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 *non-interference* that eliminate the two sources of malfunction.

Definition 4 *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 5 *Two production rules $Bu \rightarrow x\uparrow$ and $Bd \rightarrow x\downarrow$ are said to be non-interfering in a computation if and only if $\neg Bu \vee \neg Bd$ is an invariant of the computation. A production-rule set is non-interfering if every pair of complementary production rules in the set is non-interfering.*

3.2 Deterministic Execution

In any state, any subset of the set of production rules with a true guard, so-called *enabled* PRs, can fire concurrently. This execution model is difficult to work with since it requires us to consider all possible concurrent executions in each state. Fortunately, under stability and non-interference, the concurrent firings of any two PRs is equivalent to the firing of the two rules in any order.[3] In other words, a stable and non-interfering QDI computation is *deterministic*. An early study of determinism in speed-independent circuits is in [23].

Theorem 1 *Any concurrent execution of a stable and non-interfering 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 *semi-modularity*[37], and *persistency*[11].

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

[2] As already mentioned, we make a minor exception to the non-interference rule when we add vacuous transitions to implement state-holding elements, and that is the reason why some inverters are "weak."

[3] This property is sometimes called the *diamond property*, or *strong confluence*, or *Church-Rosser* and plays an important role in the lambda-calculus theory of programming languages and rewrite systems.

3.3 Cycles and Self-Invalidating Transitions

Since hardware computations are non-terminating, each transition $z\uparrow$ is followed, after a number of other transitions, by transition $z\downarrow$, and vice versa. Since the guards Bu and Bd of those transitions are mutually exclusive, the chain of transitions betwen $z\uparrow$ and $z\downarrow$ must contain a transition that invalidates Bu. Hence, transition $z\uparrow$ invalidates itself through a sequence of intermediate transitions. Can we still say that Bu is stable? Is it possible that the effect of $z\uparrow$ propagates through the cycle of gates fast enough to invalidate itself? At the electrical level, could the voltage $V(z)$ stabilize at an intermediate value close to $Vdd/2$?

Arguing that such a ring of operators is not self-invalidating is equivalent to arguing that the ring oscillates. This is an electrical property of the circuit that relates the slew rates of transitions, the gain of the operators, and the number of operators on the ring. We require that any cycle of operators be implemented with a number of stages at least equal to a chosen minimum to guarantee that the cycle is not self-invalidating. (In CMOS, three restoring operators with good gain are usually sufficient, although we usually require five to be safe.)

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*). The direct implementation of the rule is a ring containing just one operator. *We exclude self-invalidating production rules since, in most implementations, they would violate the stability condition.*

3.4 Transition Ordering, Acknowledgment, and Isochronic Fork

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.

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 following example in which the specification of the computation requires an ordering of transitions on variables x and y defined by the sequence:

$$\ldots\ x\uparrow;\ \ y\uparrow;\ \ \ldots; x\uparrow;\ \ z\uparrow\ldots$$

Implementing this sequence of transitions requires introducing at least one control variable c to distinguish the states in which $x\uparrow$ causes $y\uparrow$ from the states in which $x\uparrow$ causes $z\uparrow$, leading to PRs of the form:

$$\begin{aligned} c \wedge x &\rightarrow y\uparrow \\ \neg c \wedge x &\rightarrow z\uparrow \end{aligned}$$

As shown in Figure 3, x as the output of gate Gx 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*[30]. (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 Figure 15 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 multi-transition path. If the class of DI circuits is too small to include all interesting computations, is the QDI class large enough? It can be proved by constructing a Turing machine as a QDI circuit (see [26]) that the class of QDI circuits is Turing complete, i.e., all Turing-computable functions have a QDI implementation.

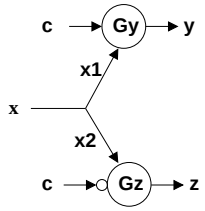

Figure 3: The fork (x,x1,x2) is isochronic: a transition on x1 causes a transition on y only when c is true, and a transition on x2 causes a transition on z only when c is false. Hence, certain transitions on x1 and on x2 are not acknowledged, and therefore a timing assumption must be used to guarantee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

3.5 Summary

The synthesis method produces circuits that are stable and non-interfering and therefore hazard-free, except for isochronic-fork hazards. Isochronic forks have to satisfy a timing assumption, and rings of operators must contain a minimal number of restoring operators. The conditions on isochronic forks and operator rings can always be satisfied by adding restoring delay elements (inverters).

4 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 $p2$ 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.

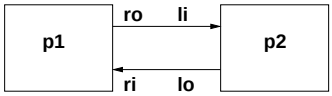

Figure 4: Implementation of a "bare" channel (L, R) with two handshake wires: (lo, ri) and (ro, li).

4.1 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 Figure 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** (zero).

4.1.1 Two-phase Handshake

The simplest handshake protocol implementing the slack-0 communication between R and L is the so-called *two-phase handshake* protocol, also called *non-return to zero* (NRZ). The protocol is defined by the following handshake sequence Ru for R and Lu for L:

Ru : $ro\uparrow$; [ri]
Lu : [li]; $lo\uparrow$

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-0 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

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

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 (upgoing) and odd (downgoing) 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:

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

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.

4.1.2 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:

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

In this case, the only possible interleaving of transitions (we cannot strictly order the waits) 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-0 communication between R and L. It can even be argued that this implementation is in fact the sequencing of two slack-0 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*.

4.2 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 Figure 5. The protocols are as follows:

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

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

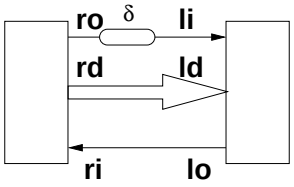

Figure 5: A bundled-data communication protocol. The cigar shape on the control wire (ro, li) indicates that the delay δ on the wire has been adjusted to be longer than the delays on the data wires.

4.3 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.

4.4 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 Figure 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

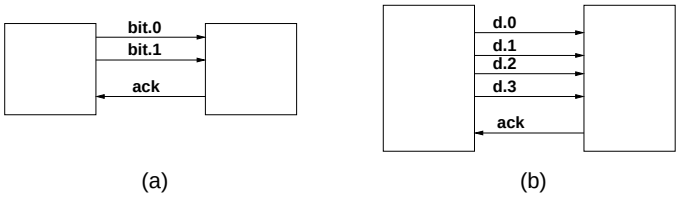

Figure 6: (a) A dual-rail coding of a boolean data-channel; (b) A 1-of-4 coding of a 4-valued integer data channel. Observe that no delay element is needed.

4.5 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 4 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 Figure 6(b).)

4.6 k-out-of-N Codes

The one-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 one-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.

4.7 Berger Codes

Berger codes are constructed differently. Let X be the code word of data word D; X is the concatenation of the unchanged data word D and a code field W, such that W is the binary representation of the number of zeros in D. The validity test is defined as:

$$v(X) \equiv (\#\text{zeros in } D = \text{dec}(W))$$

where $\text{dec}(W)$ is the decimal value of W. The following table gives the Berger codes for four data words of size 4—each column represents a code word, with the most significant bit of W at the bottom:

D	0	1	0	1
	0	0	1	1
	0	0	0	1
	0	0	0	1
W	0	1	1	0
	0	1	1	0
	1	0	0	0

The following is an informal argument of why Berger codes are DI. Since the neutral value consists of all zeros, any intermediate code word (Di, Wi) has more zeros than the final value (D, W), either in Di or in Wi or both. In all three cases, the number of zeros represented by Wi is less than the number of zeros in Di. Therefore no intermediate code word is valid.

4.8 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. Secondly, the coding and decoding of a data word must be simple. Thirdly, 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 partitionings and for the one-of-4 code for partitionings down to a quarter-byte. For all those practical reasons, dual-rail and 1-of-4 are used almost exclusively in asynchronous VLSI design.

4.9 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:

$$\begin{array}{ll} Send: & data\Uparrow;\ [v(ack)];\ data\Downarrow;\ [n(ack)] \\ Receive: & [v(data)];\ ack\uparrow;\ [n(data)];\ ack\downarrow \end{array}$$

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.

4.9.1 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 inpout 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:

$$\begin{array}{ll} Send\ (passive): & [v(ack)];\ data\Uparrow;\ [n(ack)];\ data\Downarrow \\ Receive\ (active): & ack\uparrow;\ [v(data)];\ ack\downarrow;\ [n(data)]\ . \end{array}$$

4.9.2 Example: One-bit Channel

A one-bit channel between a sender and a receiver is implemented as in Figure 7. Two data wires $(r1, l1)$

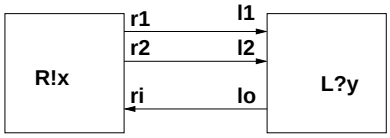

Figure 7: Handshake wires for a one-bit DI data channel.

and $(r2, l2)$ 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 $l1 \vee l2$ and $\neg l1 \wedge \neg l2$. For an active send and a passive receive, we get

$$\begin{array}{ll} R!x: & [x \longrightarrow r1\uparrow [] \neg x \longrightarrow r2\uparrow];\ [ri];\ r1\downarrow, r2\downarrow;\ [\neg ri] \\ L?y: & [l1 \vee l2];\ [l1 \longrightarrow y\uparrow [] l2 \longrightarrow y\downarrow];\ lo\uparrow;\ [\neg l1 \wedge \neg l2];\ lo\downarrow\ . \end{array}$$

In the send, the selection $[x \rightarrow r1\uparrow [] \neg x \rightarrow r2\uparrow]$ assigns the value of x to the data wires of port R. In the receive, the selection $[l1 \rightarrow y\uparrow [] l2 \rightarrow y\downarrow]$ assigns the 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 $[l1 \vee l2]$ is superfluous, since the selection following it also includes waiting for $l1$ or $l2$. We can rewrite the HSE of the receive as

$$L?y: \quad [l1 \longrightarrow y\uparrow [] l2 \longrightarrow y\downarrow]; \; lo\uparrow; \; [\neg l1 \wedge \neg l2]; \; lo\downarrow \; .$$

The solution for passive send and active receive is

$$\begin{array}{ll} R!x: & [ri]; \; [x \longrightarrow r1\uparrow [] \neg x \longrightarrow r2\uparrow]; \; [\neg ri]; \; r1\downarrow, r2\downarrow \\ L?y: & lo\uparrow; \; [l1 \longrightarrow y\uparrow [] l2 \longrightarrow y\downarrow]; \; lo\downarrow; \; [\neg l1 \wedge \neg l2] \; . \end{array}$$

5 Basic Building Blocks: Sequencing, Storage, Computation

The three basic building blocks are (1) a circuit that sequences two bare communication actions—we shall see that 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. Those circuits will be derived in some details from their HSE specifications, in order to give the reader some understanding of the logic synthesis method that produces the circuits.

5.1 Sequencer

The basic sequencing building block is the "sequencer" process, also called "left-right buffer":

$$p1: \quad *[L; \; R]$$

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*. The most general specification of the environment is

$$env: \quad *[L'] \parallel *[R']$$

where L' connects to L and R' connects to R.

The implementations vary significantly depending on whether the two ports are active or passive. The simplest 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

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

5.1.1 Logic Synthesis: From HSE to PRS

Next, the HSE of $p1$ is implemented as a production rule set that can then be realized as a CMOS transistor network. The HSE is now the specification of the circuit. This step is the core of the logic synthesis, since it requires implementing the partial order of transitions explicitly. (Several alternative logic synthesis algorithms exist, for instance[12].) By a simple observation of the HSE, we are encouraged to try the following PR set as a first attempt:

$$\begin{array}{rcl} \neg ri & \rightarrow & lo\uparrow \\ li & \rightarrow & lo\downarrow \\ \neg li & \rightarrow & ro\uparrow \\ ri & \rightarrow & ro\downarrow \; . \end{array}$$

While it is indeed possible to fire the PRs in the order of HSE, namely top to bottom, other firing orders are possible that do not correspond to the specification. In particular, $\neg li \rightarrow ro\uparrow$ can fire in the initial state, which would violate the HSE specification. In order to prevent PR $G \rightarrow t$ from firing in state s characterized by the condition S on the variables of the system, we strengthen the guard G with the simplest condition S' such that $G \wedge S' \Rightarrow \neg S$, i.e. the PR becomes $G \wedge S' \rightarrow t$. If we try this approach to prevent PR $\neg li \rightarrow ro\uparrow$

from firing in the initial state, we conclude that we cannot. The reason is that 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 need to introduce a *state variable* to distinguish those two states. This phase of the synthesis is usually called *state assignment*. In this example, a single state variable x (initially false) suffices:

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

Now, all the states that need to be distinguished are uniquely determined and we can generate a PR set that implements the HSE:

$$\begin{aligned} \neg x \wedge \neg ri &\rightarrow lo\uparrow \\ li &\rightarrow x\uparrow \\ x &\rightarrow lo\downarrow \\ x \wedge \neg li &\rightarrow ro\uparrow \\ ri &\rightarrow x\downarrow \\ \neg x &\rightarrow ro\downarrow \end{aligned}$$

The previous PR set is implementable, but it contains only state-holding gates. An additional transformation called *guard symmetrization* is usually applied, which consists of adding vacuous transitions so as to make the guards of the PRs of a gate boolean complements of each other. This leads to the two solutions shown in Figure 8. In the first implementation, the set-reset gate of x is replaced with a C-element $x = (li\ \underline{C}\ ri)$. In the second one, the set-reset is implemented with cross-coupled nor-gates.

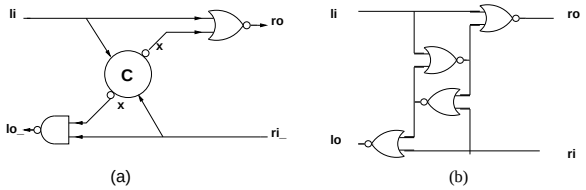

Figure 8: Implementation of an Active-Active Buffer (sequencer): (a) with a C-element implementation of the state bit, (b) with a cross-coupled nor-gate implementation of the state bit.

5.1.2 Q-Element and Passive-Active Buffer

From the active-active buffer, we derive a slightly different process, called a "Q-element," with the specification:

$$p2:\quad *[[\overline{L}];\ R;\ L]$$

This process has the same two ports as $p1$, but sequences the actions differently: $p2$ first waits until there is a pending communication on L from the environment, then it first does R and finally does L. Its circuit implementation is used in many control structures.

For bare ports L and R, and choosing L passive and R active, the HSE of $p2$ is

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

which is the HSE of the active-active buffer started in the state following $lo\downarrow$, and with variable li inverted. (For instance the nor-implementation of the AA buffer with xt and xf initialized to true and false respectively, and with li inverted implements the Q-element.)

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 Figure 9.

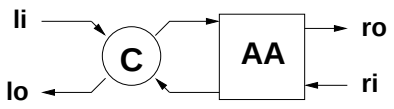

Figure 9: A passive-active buffer implemented as an active-active buffer with a C-element as an active-to-passive converter on port L.

5.1.3 Reshuffling and Half-Buffers

We have already mentioned that the downgoing phase of a four-phase handshake is solely for the purpose of resetting all variables to their initial (neutral state) values, usually false. The designer therefore has some leeway in the sequencing of the downgoing actions of a communication with respect to other actions of a HSE. The transformation that moves a part of a handshake sequence in a 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.

5.1.4 Simple Half-Buffer

A rather drastic reshuffling is the following one:

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

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 Figure 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*, more specifically a *simple half-buffer* (SHB). (The term half-buffer was introduced by Andrew Lines in [25].) 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 Figure 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 David Muller[37].

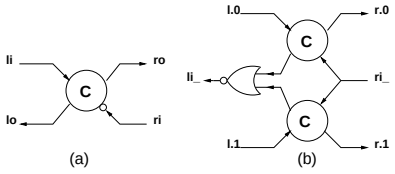

Figure 10: A simple half-buffer: (a) bare handshake, (b) with one bit of data transmitted from L to R.

5.1.5 C-Element Full Buffer

Another (less drastic) reshuffling of the original HSE is:

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

The PRS is:

$$\begin{aligned}
\neg ro \wedge li &\rightarrow lo\uparrow \\
lo \wedge \neg ri &\rightarrow ro\uparrow \\
ro \wedge \neg li &\rightarrow lo\downarrow \\
\neg lo \wedge ri &\rightarrow ro\downarrow
\end{aligned}$$

leading to the two C-element implementation of Figure 11. Since ri is false in the neutral state of R, the

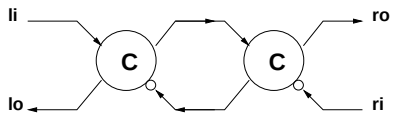

Figure 11: An implementation of a bare-handshake full-buffer.

HS sequence of L can complete without the environment of R being started, i.e. even if ri doesn't 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 Figure 12.

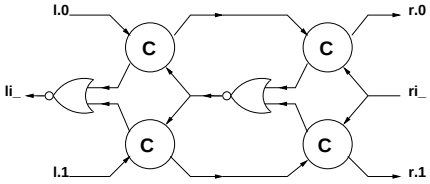

Figure 12: A full-buffer FIFO stage transmitting one bit of data from left to right

5.2 Reshuffling and Slack

In most cases, 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. But at the same time, reshuffling may 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. As an illustration, Figure 13 shows different reshufflings of $*[L;R]$, starting with the maximal, slack-0, reshuffling which reduces the implementation to two wires, and ending with the non-reshuffled (maximally decoupled) slack-1 implementation, which is also the most complex.

5.3 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. If we assume that the read and write requests from the environment are mutually exclusive, which requires that the channels connecting the register to the environment have no slack, then the probe construct can be used to select a read or write request, and the code for the register is

$$\begin{aligned}
*[[&\overline{P} \longrightarrow P?x \\
\,[]\,&\overline{Q} \longrightarrow Q!x \\
]]\ .&
\end{aligned}$$

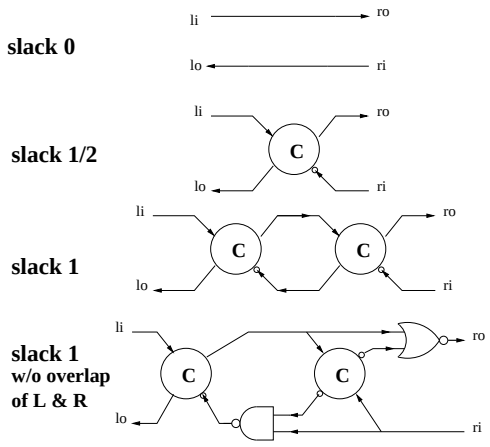

Figure 13: Tradeoff between reshuffling and slack in the implementation of the buffer control $*[L; R]$

If the probe cannot be used, for instance because the channels cannot maintain the order of the read/write requests, then an extra control port, say C, is needed to carry the read/write information, as in the following program:

$$
\begin{array}{l}
*[C?c; \\
\quad [c \ \longrightarrow \ P?x \\
\quad []\neg c \ \longrightarrow \ Q!x \\
\quad]] \ .
\end{array}
$$

Although the solution with control channel is slighty more complicated, it has the advantage that the channels can accept any amount of slack without changing the correctness of the system. We say that the solution is *slack elastic*. Slack elasticity is an important property, because it allows more freedom in the decomposition of large computations into a collection of communicating modules, and it also allows slack matching for performance optimization. If we were to try and add slack to the first solution, the order of read and write requests would be lost.

Let us implement the first solution. The handshaking expansion uses dual-rail encoding. As shown in Figure 14, input port P is implemented with two input wires, $pi1$ for receiving the value **true**, and $pi2$ for receiving the value false; and one acknowledge wire, po. Output port Q is implemented with two output wires, $qo1$ for sending the value **true**, and $qo2$ for sending the value **false**; and one request wire, qi. Variable x is also dual-rail encoded as the pair of variables xt, xf. The handshaking expansion gives:

$$
\begin{array}{lll}
*[[pi1 & \longrightarrow & xf\downarrow; \ xt\uparrow; \ po\uparrow; \ [\neg pi1]; \ po\downarrow \\
\quad []pi2 & \longrightarrow & xt\downarrow; \ xf\uparrow; \ po\uparrow; \ [\neg pi2]; \ po\downarrow \\
\quad []xt \wedge qi & \longrightarrow & qo1\uparrow; \ [\neg qi]; \ qo1\downarrow \\
\quad []xf \wedge qi & \longrightarrow & qo2\uparrow; \ [\neg qi]; \ qo2\downarrow \\
\quad]] \ . & &
\end{array}
$$

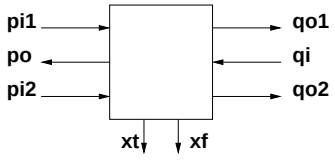

Figure 14: Handshake wires for the single-bit register

5.3.1 Writing an Asynchronous Register

The PRs for the write part of the register (the first two lines of the HSE) are:

$$\begin{array}{rcl@{\qquad}rcl} pi1 & \rightarrow & xf\downarrow & pi2 & \rightarrow & xt\downarrow \\ \neg xf & \rightarrow & xt\uparrow & \neg xt & \rightarrow & xf\uparrow \\ pi1 \wedge xt & \rightarrow & po\_\downarrow & pi2 \wedge xf & \rightarrow & po\_\downarrow \\ \neg pi1 & \rightarrow & po\_\uparrow & \neg pi2 & \rightarrow & po\_\uparrow \end{array}$$

(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 multi-bit 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:

$$\begin{array}{rcl} (pi1 \wedge xt) \vee (pi2 \wedge xf) & \rightarrow & po\_\downarrow \\ \neg pi1 \wedge \neg pi2 & \rightarrow & po\_\uparrow \end{array}$$

A direct CMOS implementation, usually preferred, is shown in Figure 15(a). A pass-transistor implementation is shown in Figure 15(b).

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. The cost is high since a *wack* is needed for every bit of storage.

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.

5.3.2 Reading an Asynchronous Register

The read-part of the register is simple. The PRs are

$$\begin{array}{rcl} xt \wedge qi & \rightarrow & qo1\uparrow \\ \neg qi & \rightarrow & qo1\downarrow \\ xf \wedge qi & \rightarrow & qo2\uparrow \\ \neg qi & \rightarrow & qo2\downarrow \, . \end{array}$$

Those PRs can be implemented directly in CMOS. If the environment does not modify x (through a write) until the end of the four-phase read handshake, the second and fourth rules can be implemented as

$$\begin{array}{rcl} \neg xt \vee \neg qi & \rightarrow & qo1\downarrow \\ \neg xf \vee \neg qi & \rightarrow & qo2\downarrow \end{array}$$

which leads to a two nand-gate implementation. Both implementations are shown in Figure 15.

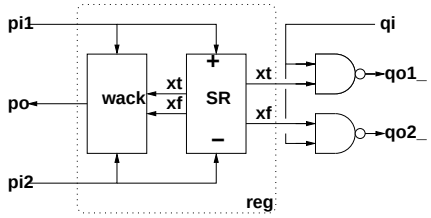

Figure 15: An implementation of the single-bit register: The write-acknowledge (wack) can be implemented with standard gates, or with a single complex gate, or with pass-transistors; SR is implemented either with cross-coupled nor-gates or cross-coupled inverters.

5.4 N-bit Register and Completion Tree

The generalization to an n-bit register is straightforward: The 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 $y = x1 \underline{C} x2 \underline{C} ... \underline{C} xn$ follows the restricted protocol in which a transition on the output y is always preceded by exactly one transition on each input, as follows:

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

In this case, the n-input C-element can be decomposed into a binary tree of 2-input C-elements without the danger of introducing unstable transitions on the intermediate variables introduced by the decomposition. Such a C-element tree is called a *completion tree*[33].

The completion tree puts a delay proportional to $logn$ 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 forked to all bits of the register. The n-bit register is shown in Figure 16.

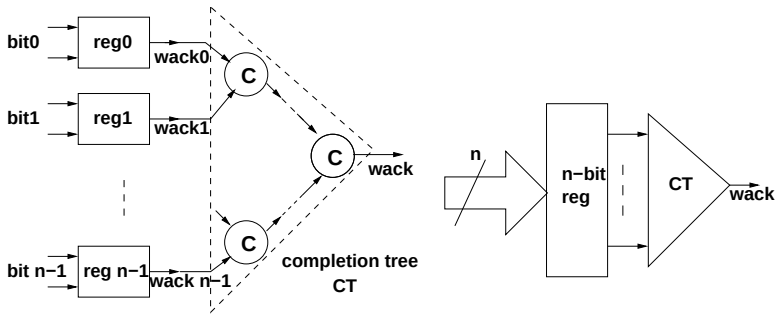

Figure 16: An n-bit register as the composition of n single-bit registers. The global write-acknowledge signal is generated by a completion tree combining the single-bit write-acknowledges.

5.5 Function Evaluation

Computation is done by evaluating a function f for a given value of its parameters. Without loss of generality, consider the computation evaluating the boolean function $f(X)$ for the current value of parameter X and assigning the result to Y, i.e., implementing $Y := f(X)$. The canonical handshake for such a (one-bit) computation is

$$F(X, Y) \equiv *[[v(X)]; [f(X) \longrightarrow y1\uparrow [] \neg f(X) \longrightarrow y0\uparrow]; [n(X)]; y0\downarrow, y1\downarrow] \ .$$

Input parameter X and result Y are both coded in a DI code (in the above example in dual-rail), and both input parameters and results go through the neutral/valid/neutral cycle: $v(X)$ and $n(X)$ are the validity and neutrality tests for the input X; output Y is set to a valid value corresponding to the value of $f(X)$ and then reset to the neutral value. It is important to observe that the above handshake is not a complete four-phase handshake on the ports X and Y and that F is not the handshake of a process, unless we consider that X and Y together form a port. The most general environment is specified by the HSE

$$env(F) \equiv *[X \Uparrow; [v(Y)]; X \Downarrow; [n(Y)]] \ ,$$

where $X \Uparrow$ represents setting X to a valid value, and $X \Downarrow$ represents setting X to the neutral value. Although the HSE of the function is not a process, it is composable. If $f = h \circ g$, $F(X,Y) = (G(X, Z) \parallel H(Z, Y))$. It can be proved that F can be directly implemented by the PR set:

$$
\begin{aligned}
v(X) \wedge f0(X) &\rightarrow y0\uparrow \\
v(X) \wedge f1(X) &\rightarrow y1\uparrow \\
n(X) &\rightarrow y0\downarrow \\
n(X) &\rightarrow y1\downarrow
\end{aligned}
$$

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

5.5.1 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

$$
\begin{aligned}
&[(a0 \wedge b0) \vee (a1 \wedge b1) \longrightarrow y1\uparrow \\
&[](a0 \wedge b1) \vee (a1 \wedge b0) \longrightarrow y0\uparrow \\
&] .
\end{aligned}
$$

Observe that (1) the two guards of the coded version are no longer boolean complements of each other, and all negations have disappeared; and (2) in this example, each guard of the dual-rail function evaluation implies the validity of both inputs. Hence, the PRS can be simplified as:

$$
\begin{aligned}
(a0 \wedge b0) \vee (a1 \wedge b1) &\rightarrow y1\uparrow \\
(a0 \wedge b1) \vee (a1 \wedge b0) &\rightarrow y0\uparrow \\
\neg a1 \wedge \neg a0 \wedge \neg b1 \wedge \neg b0 &\rightarrow y1\downarrow \\
\neg a1 \wedge \neg a0 \wedge \neg b1 \wedge \neg b0 &\rightarrow y0\downarrow
\end{aligned}
$$

Even for this simple function, the neutrality tests (the guards of the last two PRs) requires four p-transistors in series, which is unacceptable in today's technology.

5.5.2 Precharge Function Evaluation

We are going to decouple the validity/neutrality test from the function evaluation in order to simplify the reset condition for the function. Consider the HSE for the function F, in which we have introduced a variable v that is assigned the result of the validity and neutrality tests:

$$F(X, Y) \equiv *[[v(X)]; v\uparrow; [f1(X) \longrightarrow y1\uparrow [] f0(X) \longrightarrow y0\uparrow]; [n(X)]; v\downarrow; y0\downarrow, y1\downarrow] .$$

The above HSE is equivalent to the parallel composition of the two following HSEs:

$$VN \equiv *[[v(X)]; v\uparrow; [n(X)]; v\downarrow] ,$$

and

$$PCF(X, Y) \equiv *[[v \wedge f1(X) \longrightarrow y1\uparrow [] v \wedge f0(X) \longrightarrow y0\uparrow]; [\neg v]; y0\downarrow, y1\downarrow] .$$

The first one, VN, computes the validity/neutrality tests. It seems that we have just moved the difficulty from one place to another. But because of the symmetry of VN, the tests can now be decomposed into a completion tree satisfying the proper limitations on transistor chains. The second one, PCF, computes the function F as a PCF with v as a control signal. The outputs $y0\_$ and $y1\_$ are usually inverted through a staticizer.

As an example, let us return to the equality function. With a separate circuit computing the validity/neutrality signal v, the production rules of the function are

$$
\begin{aligned}
v \wedge ((a0 \wedge b0) \vee (a1 \wedge b1)) &\rightarrow y1\_\downarrow \\
v \wedge ((a0 \wedge b1) \vee (a1 \wedge b0)) &\rightarrow y0\_\downarrow \\
\neg v &\rightarrow y1\_\uparrow \\
\neg v &\rightarrow y0\_\uparrow .
\end{aligned}
$$

The circuit implementation is shown in Figure 17.

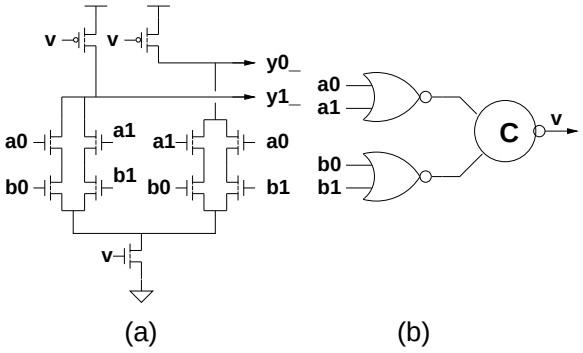

Figure 17: Precharge implementation of the boolean-equality function. (a) Precharge function evaluation; (b) validity/neutrality test circuit. For large input, the single C-element is replaced with a completion tree.

6 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 [33], and used in the design of the first asynchronous microprocessor[31].)

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[62]. This approach was introduced for the design of the MiniMIPS[32].

6.1 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 Figure 18. The control part implements L as active and R as passive, leading 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 Figure 19. For the send part (the function evaluation), the implementation is the same whether R is active or passive. Therefore, when this scheme is used, the input ports are usually implemented as active and the output ports as 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 P, 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.

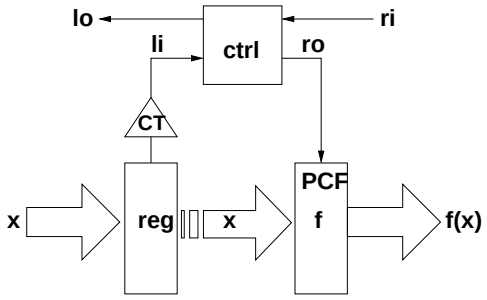

Figure 18: The control-data decomposition technique applied to a simple buffer stage. In this case, the input port is active and the output port passive

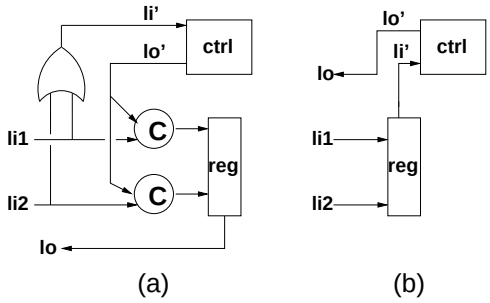

Figure 19: Implementation of a one-bit input interface for (a) a passive port, (b) an active port

6.2 Precharge Buffers and 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. Secondly, 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 re-shuffled 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).

6.3 Simple 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 the HS on R before completing the HS on L

$$*[[re];\ [f0(L) \longrightarrow r0\uparrow [] f1(L) \longrightarrow r1\uparrow];\ le\downarrow;\ [\neg re];\ r0\downarrow, r1\downarrow;\ [\neg l0 \wedge \neg l1];\ le\uparrow]\ .$$

The above HSE can be decomposed into two parallel components: one computing the function as a standard precharge function block, PCF, the other one, LHS, completing the left (input port) HS by computing le, and in the general case, the internal enable signal en, as follows:

$$PCF \equiv *[[re \wedge le];\ [f0(L) \longrightarrow r0\uparrow [] f1(L) \longrightarrow r1\uparrow];\ [\neg re \wedge \neg le];\ r0\downarrow, r1\downarrow]$$

$$LHS \equiv *[[v(L) \wedge v(R)];\ le\downarrow;\ [\neg v(L) \wedge \neg v(R)];\ le\uparrow]$$

In LHS, $v(L)$ and $v(R)$ are the validity/neutrality conditions for ports L and R. Because we use only one-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:

$$\begin{aligned} re \wedge le \wedge f0(L) &\rightarrow r0\uparrow \\ re \wedge le \wedge f1(L) &\rightarrow r1\uparrow \\ \neg le \wedge \neg re &\rightarrow r0\downarrow, r1\downarrow \end{aligned}$$

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 Figure 20. We leave it as an exercise to the reader to check that the PCHB

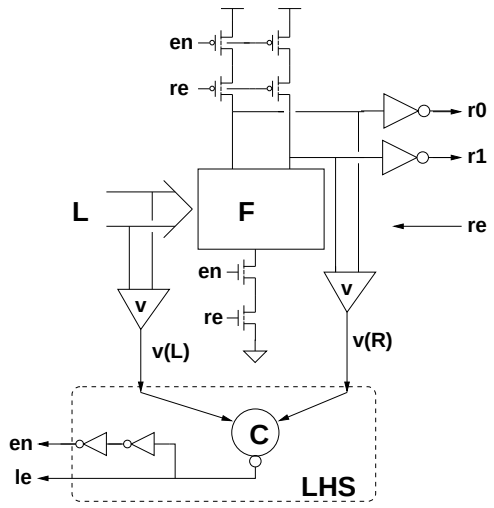

Figure 20: Implementation of a simple pipeline stage as a precharge half-buffer. Signal en has been introduced for the sake of generality. In this case, it is identical to le, but not in general.

reshuffling is indeed a half-buffer by checking that the sequential composition of two PCHBs is a full buffer, i.e., the left-most handshake can terminate before the right-most handshake starts.

The PCHB has short forward latency—only two elementary transitions, and pipelines composed of PCHBs can have excellent throughput (an average of 18 elementary transitions for the MiniMIPS[25].)

6.4 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 $L_1, ...L_m$ is the output of a precharge function block
$$r_j = PCF(F_j(L_1, ..., L_m), en_R, re) \ ,$$
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 left-enable $l_i.e$ is the (inverted) 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, ...R_k depending on L_i, we have
$$l_i.e\_ = v(L_i) \ \underline{C} \ v(R_1) \ \underline{C} \ v(R_2) ... \ \underline{C} \ v(R_k) \ .$$
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 Figure 21(a).)

3. The internal enable signal en_R is the C-element combination of all left-enable signals $l_i.e$:
$$en_R = l_1.e \ \underline{C} \ l_2.e ... \ \underline{C} \ l_m.e$$

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 $c0$, $c1$ where $c0$ indicates that L is used, and $c1$ that it is not used. The computation of the left enable le is then done as in Figure 21(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.

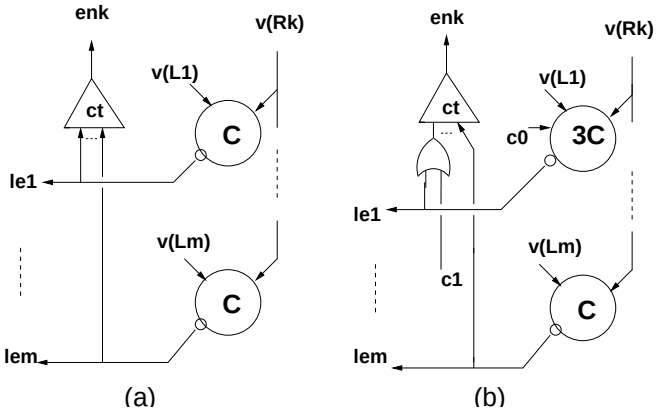

Figure 21: Left-enable computation for a general PCHB scheme: (a) LHS when all inputs needed for R_k are unconditional; (b)LHS when one input L_i is conditional. Control signals c0 ("L_i is used") and c1 ("L_i is not used") may have to be generated as extra outputs.

6.5 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.

6.5.1 Half-Buffer Controlled Merge

A two-way controlled merge is a process that 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

$$
\begin{array}{lll}
*[C?c; [c0 & \longrightarrow & R!(L?) \\
\quad [] c1 & \longrightarrow & R!(M?) \\
\ ] \quad\ ] & . &
\end{array}
$$

For boolean ports L, M, and R, the circuit is shown in Figure 22. 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

$$
\begin{array}{rcl}
ce \wedge re \wedge (c0 \wedge l0 \vee c1 \wedge m0) & \rightarrow & r0\uparrow \\
ce \wedge re \wedge (c0 \wedge l1 \vee c1 \wedge m1) & \rightarrow & r1\uparrow \\
\neg re \wedge \neg ce & \rightarrow & r0\downarrow, r1\downarrow .
\end{array}
$$

The equations for the left enables and internal enable are as follows:

$$
\begin{array}{rcl}
le\_ & = & v(L)\ \underline{C}\ v(R)\ \underline{C}\ c0 \\
me\_ & = & v(M)\ \underline{C}\ v(R)\ \underline{C}\ c1 \\
ce & = & le \vee me
\end{array}
$$

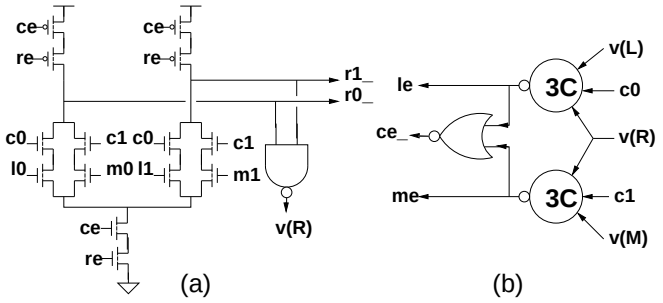

Figure 22: A two-way controlled merge implemented as a PCHB: (a) Precharge function, (b) LHS.

6.5.2 Half-Buffer Controlled Split

The two-way controlled split is a process that receives data on one input port L and sends it on one of two output ports M and R. Which output port is selected is determined by the value received on control port C

$$
\begin{array}{lll}
*[C?c, L?x; & & \\
\quad [c0 & \longrightarrow & M!x \\
\quad [] c1 & \longrightarrow & R!x \\
\ ] \ ] & . &
\end{array}
$$

For boolean ports L, M, and R, the circuit is shown in Figure 23. This is a simple case of conditional output since exactly one of the two outputs is always used. The computed precharge function is

$$
\begin{array}{rcl}
me \wedge en \wedge c0 \wedge l0 & \rightarrow & m0\uparrow \\
me \wedge en \wedge c1 \wedge l0 & \rightarrow & r0\uparrow \\
re \wedge en \wedge c0 \wedge l1 & \rightarrow & r1\uparrow \\
re \wedge en \wedge c1 \wedge l1 & \rightarrow & m1\uparrow \\
\neg re \wedge \neg en & \rightarrow & r0\downarrow, r1\downarrow \\
\neg me \wedge \neg en & \rightarrow & mo\downarrow, m1\downarrow
\end{array}
$$

The equations for LHS are as follows:

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

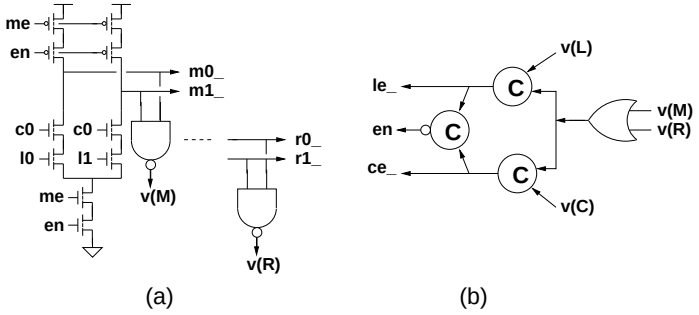

Figure 23: A two-way controlled split implemented as a PCHB. Only one of the two function blocks is shown, with (a) Precharge function block, (b) LHS.

6.5.3 Slack-Zero Split and Merge

Controlled split and merge admit relatively simple slack-zero implementations as shown in Figure 24 for two-way split and merge and for a boolean datapath. The extension to arbitrary integer datapath is straightforward.

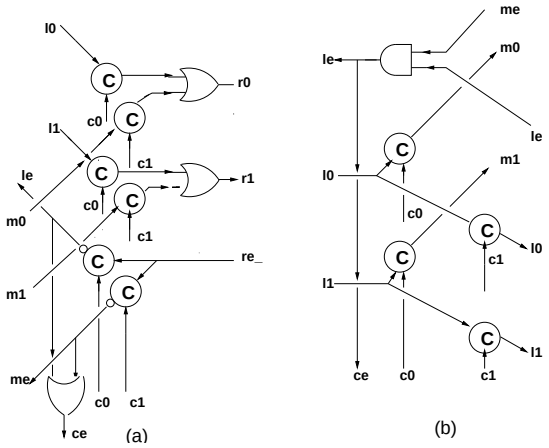

Figure 24: Slack-zero implementations of controlled merge (a) and split (b).

6.5.4 Example: Two Streams Sharing a Channel

Large data channels are often a scarce resource on 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 Figure 25.

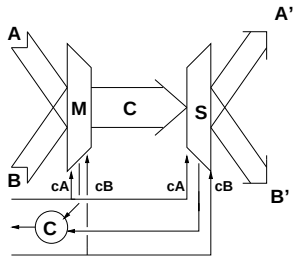

Figure 25: Two streams sharing channel C under control of dual-rail signal cA, cB.

7 Asynchronous Buses and The Curse of the Nor-Gate

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 Figure 26 for an illustration of a many-to-many bus consisting of a merge, a split, and possibly a FIFO to adjust the slack.) 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 2-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.

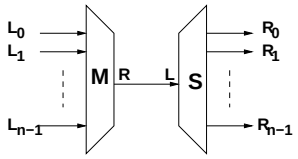

Figure 26: A many-to-many bus composed of a many-to-one merge and a one-to-many split.

7.1 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 one-of-n control input C used to select an input port, and one data output port R. As an example, let us assume that the data is dual-rail-encoded boolean. (See Figure 27(a).)

The equations for the left-enables $l_k.e$ for k from 0 to $n-1$, and for the internal enable are as follows:

$$l_k.e\_ \quad = \quad v(L_k)\ \underline{C}\ v(R)\ \underline{C}\ ck$$

$$
\begin{aligned}
ce\_ &= (\bigvee k : 0..n-1 : l_k.e) \\
en &= ce
\end{aligned}
$$

The implementation is shown in Figure 27.

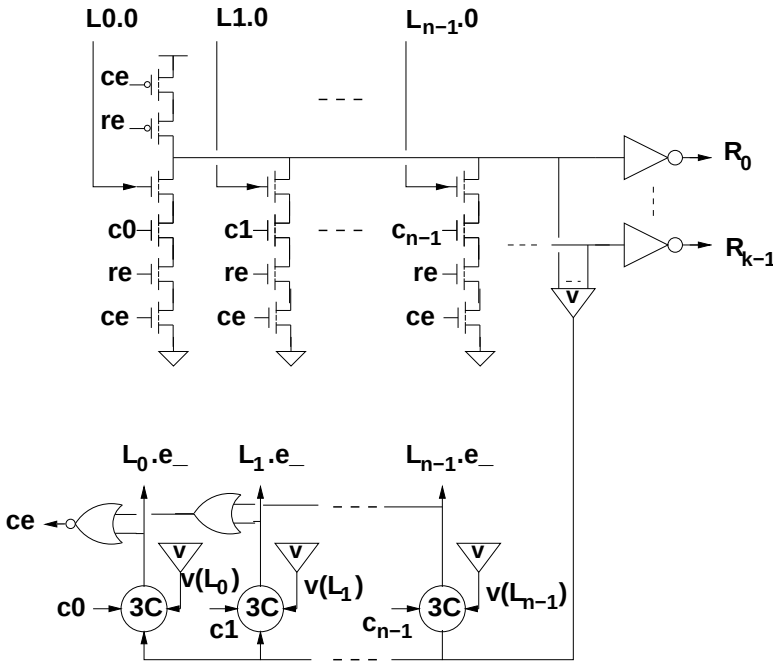

Figure 27: A PCHB implementation of a many-to-one bus. The triangles marked with a letter v are combinational gates computing the validity test of the input ports L_k and output port R. Observe that port C does not need an explicit validity test.

7.2 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_{m-1}, a one-of-m control input C used to select an output port, and one data input port L. The data is dual-rail-encoded boolean. (See Figure 28(a).)

We choose the following implementation for the left-enables and internal enable (others are possible):

$$
\begin{aligned}
ce\_ &= (\bigvee k : 0..m-1 : v(R_k)\ \underline{C}\ ck) \\
le\_ &= ce\_\ \underline{C}\ v(L) \\
en &= le
\end{aligned}
$$

The implementation is shown in Figure 28.

8 Fine-Grain High-level Decomposition

The precharge buffer templates described in the previous sections are the basic building blocks for a fine-grain, high-throughput, design style that has been used extensively in the design of complete systems, in particular the MiniMIPS, a complete QDI clone of a MIPS R3000 microprocessor designed at Caltech[32].

Since the program structure that can be fit into the buffer template is very restricted, the question arises of how to fit an arbitrary program in CHP or other high-level language into the template. This section gives

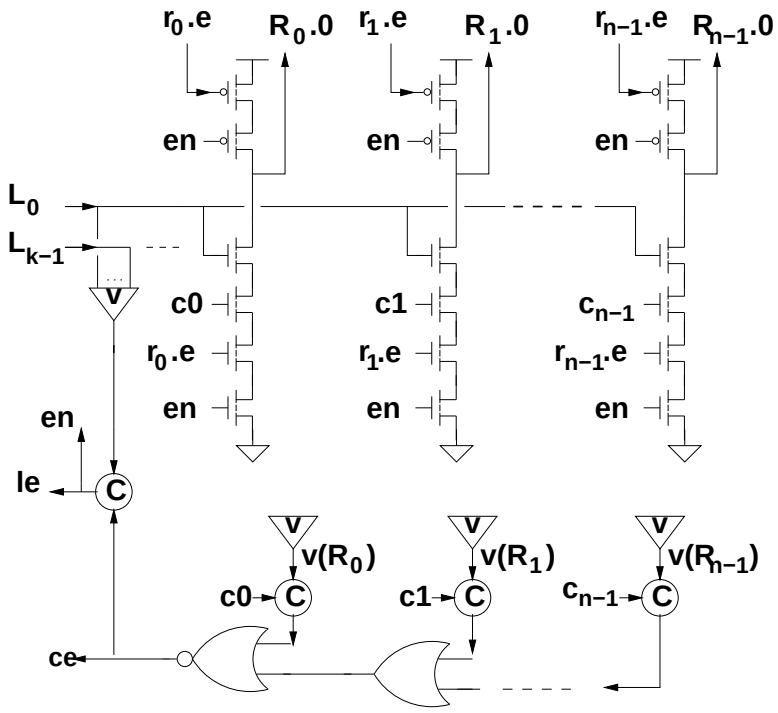

Figure 28: A PCHB implementation of a one-to-many bus. The triangles marked with a letter v are combinational gates computing the validity test of input port L and output ports R_k.

some examples of how to decompose an arbitrary program into a collection of buffers. We will not give the complete procedure—see[62].

In its most general form, a CHP process is called a "buffer" if it fulfills the following requirements:

1. Its general structure is: $P_{init}; *[P_{loop}]$
2. The initialization part P_{init} is a list of variable initializations and of constant sends. For example: $s := s0, t := t0, L!s0, M!t0$
3. The process body P_{loop} does not contain any repetition.
4. In any execution trace of either P_{init} or P_{loop}: (1) A port appears at most once; and (2) all inputs precede all outputs. (This restriction is somewhat relaxed at the implementation level by the reshuffling.)

8.1 "Phase" Transformation

Let us use an example to show how an arbitrary program is transformed to fit into the above template. Consider the CHP process:

$EX \equiv *[A?a,\ B?b;\ X!f(a,b);\ C?c;\ X!g(c);\ D?d,\ E?e;\ X!h(d,e)]$.

EX obviously does not satisfy the criteria since inputs follow outputs in the loop. But the following *phase transformation* makes it fit into the template. We first introduce a state variable s to distinguish between different "phases" of the loop, where each phase by itself satisfies the buffer criteria:

$$
\begin{array}{l}
s := 0;\ *[A?a,\ B?b;\ X!f(a,b);\ s := 1; \\
\quad\quad C?c;\ X!g(c);\ s := 2; \\
\quad\quad D?d,\ E?e;\ X!h(d,e);\ s := 0 \\
\quad]
\end{array}
$$

Then we "fold" the loop using the state variable as a guard such that each step of the loop corresponds to a phase:

$$
\begin{array}{l}
s := 0;\ *[[s = 0 \ \longrightarrow\ A?a,\ B?b;\ X!f(a,b);\ s := 1 \\
\quad\ [] s = 1 \ \longrightarrow\ C?c;\ X!g(c);\ s := 2 \\
\quad\ [] s = 2 \ \longrightarrow\ D?d,\ E?e;\ X!h(d,e);\ s := 0 \\
\quad]]
\end{array}
$$

Finally, we implement the state-variable assignments as communications with a feedback buffer. (Other implementations of state variables exist.)

$$
\begin{array}{l}
*[T?s;\ [s = 0 \ \longrightarrow\ A?a,\ B?b;\ X!f(a,b),\ S!1 \\
\quad\ [] s = 1 \ \longrightarrow\ C?c;\ X!g(c),\ S!2 \\
\quad\ [] s = 2 \ \longrightarrow\ D?d,\ E?e;\ X!h(d,e),\ S!0 \\
\quad]\qquad] \\
\\
\|\ \ T!0;\ *[S?s;\ T!s]
\end{array}
$$

Figure 29 shows the typical decomposition of a CHP process into a general precharge buffer for computation and a feedback control process—here a buffer—producing the control variable that sequences the phases.

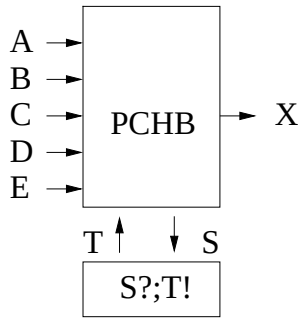

Figure 29: A decomposition of CHP process EX into a general precharge buffer for computation and a feedback control FIFO that sequences the phases.

8.2 Data-Driven Decomposition

Data-driven decomposition is a more general and more fine-grain decomposition method that identifies the independent data streams (input, computation, output) inside an arbitrary CHP program and decomposes the program accordingly into buffer templates each implementing one data stream. Using the same example, we first introduce three auxiliary channels $(X1, X)$, $(X2, X)$, and $(X3, X)$ as follows:

$$
\begin{array}{l}
\quad *[A?a,\ B?b;\ X1!f(a,b);\ C?c;\ X2!g(c);\ D?d,\ E?e;\ X3!h(d,e)]\ . \\
\|\ *[X1?x;\ X!x;\ X2?x;\ X!x;\ X3?x;\ X!x]
\end{array}
$$

Now, the first process can be decomposed into three independent buffers processes as follows:

$$
\begin{array}{l}
\quad *[A?a,\ B?b;\ X1!f(a,b)] \\
\|\ *[C?c;\ X2!g(c)] \\
\|\ *[D?d,\ E?e;\ X3!h(d,e)]
\end{array}
$$

And the second process can be made into a buffer by phase transformation:

$$
\begin{array}{l}
*[T?s;\ [s = 0 \ \longrightarrow\ X1?x;\ X!x,\ S!1 \\
\quad\ [] s = 1 \ \longrightarrow\ X2?x;\ X!x,\ S!2 \\
\quad\ [] s = 2 \ \longrightarrow\ X3?x;\ X!x,\ S!0 \\
\quad]\qquad] \\
\|\ \ T!0;\ *[S?s;\ T!s]
\end{array}
$$

Figure 30 shows the result of the transformation, which is only slightly different from the previous one: The merging of the three different outputs into one (X) takes place in a separate process in the second solution rather than in the main computation pulldown as in the first solution. (The second solution has a worse forward latency but a better throughput.) The reader must have observed that, in the case the phases always

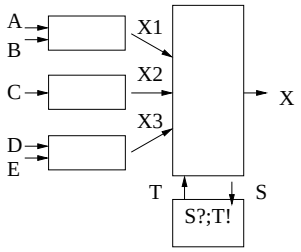

Figure 30: DDD decomposition of CHP process EX into 3 precharge buffers for computation, and a merge process controlled by a feedback control FIFO.

follow each other in the same order, the control process can be simplified. For instance in the above example, we can write

$$
\begin{array}{l}
*[\,T?s;\,[\,s=0 \;\longrightarrow\; X1?x;\;\; X!x \\
\qquad\;\; [\!]\, s=1 \;\longrightarrow\; X2?x;\;\; X!x \\
\qquad\;\; [\!]\, s=2 \;\longrightarrow\; X3?x;\;\; X!x \\
\;\;]\qquad\;\;] \\
\|\, *[\,T!0;\;\; T!1;\;\; T!2\,]\;.
\end{array}
$$

The circuit implementation is slightly more efficient.

8.3 Pipeline Vertical Decomposition

The cost of completion detection in the case of a control/data decomposed pipeline, and the cost of the validity/neutrality test in the case of fine-grain integrated pipeline are both logarithmic in the width of the datapath. Therefore implementation with a single control stage—either integrated or separate—has to be restricted to small datapaths if throughput is of any concern to the designer.

There is no strict limit to what can fit into a single module since the size depends on the function computed and the number of inputs and outputs. But the data transferred on the ports of a single PCHB is on the average no wider than a one-of-four integer. For larger datapaths, the pipeline stage is designed as the parallel composition of several buffer modules, each of them handling a portion of the datapath. Typically, a control signal c is distributed to all components by a distribution tree of cells each of which receives c on one port and sends it in parallel on a number of output ports. (See Figure 31.)

9 Arbitration

All circuits presented in the previous sections have stable and non-interfering PR sets. But, in order to implement selections in which the conditions are *not* mutually exclusive (non-deterministic 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 non-deterministic choice between two true guards.

Since stability and non-interference are equivalent to determinism, a non-deterministic computation does not admit stable and non-interfering circuit implementation. Therefore, any implementation of a non-deterministic computation contains non-monotonic transitions (glitches) that require special non-digital solutions. Those special circuits are encapsulated inside two primitive building blocks: the *arbiter* and the *synchronizer*.

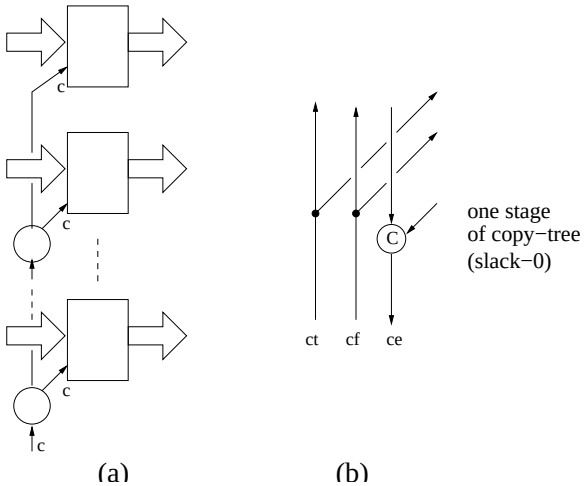

Figure 31: Vertical decomposition of a pipeline stage with wide datapath into a number of stages each handling a portion of the datapath: (a) shows the control c being sent to all partial stages through a copy tree; (b) shows a stage of the copy tree.

A fundamental, and long misunderstood, issue related to implementing a non-deterministic choice is that of *metastability*. In classical physics[4], it is impossible to put an upper bound on the time it takes for a device to make a non-deterministic 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.

9.1 Basic Arbiter

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

$$
\begin{array}{l}
arb \equiv *[[x \longrightarrow u\uparrow;\ [\neg x];\ u\downarrow \\
\qquad\ \ |\, y \longrightarrow v\uparrow;\ [\neg y];\ v\downarrow \\
\qquad]]\ ,
\end{array}
$$

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 on Figure 32. Initially, $\neg u \wedge \neg v$ holds. When either x or y or both become true, either u or v is raised but

Figure 32: The simple arbiter or mutual-exclusion element selects between two possibly concurrent inputs x and y and produces mutually exclusive outputs u and v.

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.

[4] When the problem was explained to the late Richard Feynman, he mused that metastability might be avoidable in quantum physics.

9.2 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:

$$\begin{aligned} x \wedge v\_ &\rightarrow u\_\downarrow \\ y \wedge u\_ &\rightarrow v\_\downarrow \\ \neg x \vee \neg v\_ &\rightarrow u\_\uparrow \\ \neg y \vee \neg u\_ &\rightarrow v\_\uparrow \end{aligned}$$

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 half-way 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 of time. 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 [53]. The complete nMOS circuit for the arbiter is described in [50]. A CMOS version appeared first in [29].)

(In the CMOS construction of the filter shown in Figure 33, 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

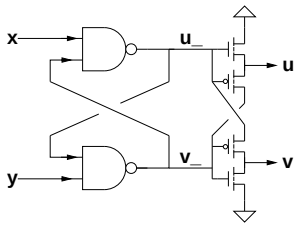

Figure 33: An implementation of the basic arbiter consisting of two cross-coupled nand-gates and a filter. The filter eliminates the spurious values of the nand-gates outputs produced during the metastable state.

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.

9.3 Channel Arbiter

If a process, say P, has to arbitrate between two input (bare) channels, say A and B, the channel arbiter $CARB$ serves as an interface to provide mutually exclusive (bare) channels A' and B' to P:

$$\begin{aligned} CARB \equiv\ & *[[\overline{A} \longrightarrow A'; A \\ & \ |\ \overline{B} \longrightarrow B'; B \\ &]] . \end{aligned}$$

Since A and B are probed, they are implemented with a passive handshake, and A' and B' with an active handshake. The direct implementation of this HSE requires at least one state variable, which can be avoided by *reshuffling* the HSE as follows:

$$
\begin{array}{l}
CARB \equiv *[[ai \longrightarrow ao'\uparrow;\ [ai'];\ ao\uparrow;\ [\neg ai];\ ao'\downarrow;\ [\neg ai'];\ ao\downarrow \\
\quad\quad | \ bi \longrightarrow bo'\uparrow;\ [bi'];\ bo\uparrow;\ [\neg bi];\ bo'\downarrow;\ [\neg bi'];\ bo\downarrow \\
\quad\quad]] \ .
\end{array}
$$

Since ai and bi can be true at the same time as indicated by the thin bar, we have to introduce an arbiter to select between ai and bi:

$$
\begin{array}{l}
arb \equiv *[[ai \longrightarrow u\uparrow;\ [\neg ai];\ u\downarrow \\
\quad\quad | \ bi \longrightarrow v\uparrow;\ [\neg bi];\ v\downarrow \\
\quad\quad]] \ .
\end{array}
$$

And we have to design a process X such that $CARB \equiv (arb \parallel X)$. Since arb replaces ai and bi with their mutually exclusive counterparts u and v, respectively, X is derived from $CARB$ by replacing ai and bi with u and v, respectively. We get:

$$
\begin{array}{l}
X \equiv *[[u \longrightarrow ao'\uparrow;\ [ai'];\ ao\uparrow;\ [\neg u];\ ao'\downarrow;\ [\neg ai'];\ ao\downarrow \\
\quad\quad [] v \longrightarrow bo'\uparrow;\ [bi'];\ bo\uparrow;\ [\neg v];\ bo'\downarrow;\ [\neg bi'];\ bo\downarrow \\
\quad\quad]] \ .
\end{array}
$$

The implementation of X is straightforward: the nand-gates are there to prevent a pending request to the arbiter to be granted before the previous four-phase handshake is completed. The combined circuits for arb and X are shown in Figure 34. The previous design can be used to arbitrate between channels with data.

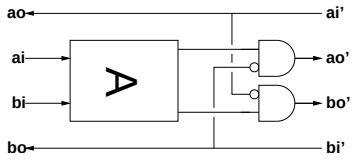

Figure 34: A slack-zero channel arbiter: The nand-gates prevent the second of two requests from proceeding before the HS of the first one is completed.

The arbiter takes as inputs a signal from each data channel A and B indicating that the channel has valid data. In the case 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.

9.4 Multiplexed Arbitration

A useful building block is an extension of the channel arbiter with a multiplexer (merge). The CHP specification of this building block is:

$$
\begin{array}{l}
XARB \equiv \ *[[\overline{A} \longrightarrow S;\ A \\
\quad\quad\quad | \ \overline{B} \longrightarrow S;\ B \\
\quad\quad\quad]] \ .
\end{array}
$$

It is obtained by combining the channel arbiter $CARB$ with a simple multiplexer (merge) $mult$ without arbitration (since A' and B' are mutually exclusive):

$$
\begin{array}{l}
mult \equiv \ *[[\overline{A}' \longrightarrow S;\ A' \\
\quad\quad\quad [] \overline{B}' \longrightarrow S;\ B' \\
\quad\quad\quad]] \ .
\end{array}
$$

We have already described the slack-zero implementation of the two-way multiplexer as a two-input or-gate and a two-way fork. The multiplexed arbiter $XARB$ is shown in Figure 35.

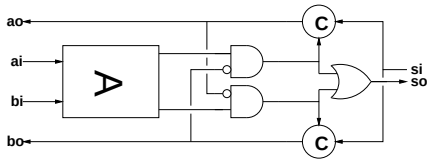

Figure 35: A multiplexed arbiter XARB composed of a channel arbiter and a simple multiplexer.

9.5 Multi-Channel Tree Arbiter

Thanks to the multiplexed arbiter, we can generalize the simple channel arbiter to the case of more than two channels. Suppose we want to implement the channel arbiter $CARB4$ which arbitrates between four channels:

$$CARB4 \equiv *[[\overline{A} \longrightarrow A \,|\, \overline{B} \longrightarrow B \,|\, \overline{C} \longrightarrow C \,|\, \overline{D} \longrightarrow D]] \ .$$

We decompose $ARB4$ into the three processes:

$$Pab \equiv *[[\overline{A} \longrightarrow AB; A \,|\, \overline{B} \longrightarrow AB; B]] \ ,$$

$$Pcd \equiv *[[\overline{C} \longrightarrow CD; C \,|\, \overline{D} \longrightarrow CD; D]] \ ,$$

$$ARB \equiv *[[\overline{AB} \longrightarrow AB \,|\, \overline{CD} \longrightarrow CD]] \ ,$$

where AB and CD are local channels. Pab and Pcd are obviously of the type $XARB$, and ARB is a bare arbiter with the corresponding input and output wires paired to form a channel. The implementation of $ARB4$ as an arbiter tree is shown in Figure 36. 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.

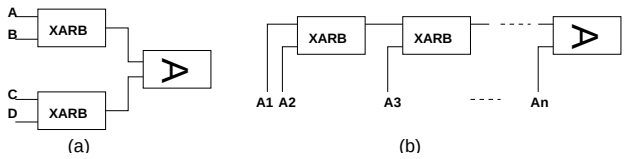

Figure 36: A 4-way tree arbiter composed of two multiplexed arbiters and a bare arbiter as the root. The generalization to an arbitrary number of channels is straightforward. The tree need not be balanced, and can be incomplete if the number of channels is not a power of two.

9.6 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 peripheral signal. A circuit to solve this problem is called a *synchronizer*.

The synchronizer has two inputs: a control input and the input signal that is to be sampled. We have chosen to specify it in such a way that it produces a dual-rail output with the two rails representing the true and false values of the sampled input. The HSE specification of the synchronizer (usually represented as on Figure 37) is:

$$\begin{aligned} sync \equiv \ *[[& re \wedge \neg x \ \longrightarrow \ r0\uparrow; \ [\neg re]; \ r0\downarrow \\ & |\, re \wedge x \ \longrightarrow \ r1\uparrow; \ [\neg re]; \ r1\downarrow \\ &]] \ , \end{aligned}$$

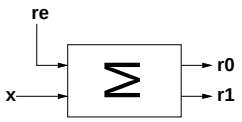

Figure 37: A synchronizer samples the value of unsynchronized input x when the control input re is high. The value read is output as the dual-rail pair (r0, r1).

where x is the input to be sampled, re is the control input, and the pair $(r0, r1)$ is the dual-rail output. When the environment raises re, the synchronizer starts evaluating x and returns the observed value by raising either $r0$ or $r1$. 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 $r1$, and similarly if x is a stable false; otherwise, the circuit asserts either $r1$ or $r0$, but not both. In practice, the "confusion interval" is a very short period of time, approximately the delay of the single inverter used to invert the input. But the synchronizer must work correctly—i.e., raise either $r1$ or $r0$—even when x changes during the confusion interval. Implementing a synchronizer properly is difficult enough that it is usually avoided.

9.7 Implementing the Synchronizer

The difficulty is that, unlike the arbiter, the synchronizer as defined by the HSE program *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 $r0\uparrow$: first, the second guard must be "locked out", so that $r1$ cannot happen; secondly, the assignment $r0\uparrow$ 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. It is clear that in order to remove the race condition, we will have to introduce intermediate *stable* variables (they remain true once evaluated to true) to stabilize the selection of one of the two guards.

We introduce variables $a0$ and $a1$ for that purpose:

$$
\begin{array}{lll}
sync1 \equiv & *[[re \wedge \neg x & \longrightarrow a0\uparrow;\ [a0];\ r0\uparrow;\ [\neg re];\ a0\downarrow;\ [\neg a0];\ r0\downarrow \\
& \mid re \wedge x & \longrightarrow a1\uparrow;\ [a1];\ r1\uparrow;\ [\neg re];\ a1\downarrow;\ [\neg a1];\ r1\downarrow \\
&]] \ . &
\end{array}
$$

We decompose $sync1$ into three parts: $int0$ and $int1$ "integrate" the possibly oscillating values of x into a stable false value $a0$ and a stable true value $a1$, and SEL selects between $a0$ and $a1$:

$$int0 \equiv *[[re \wedge \neg x \longrightarrow a0\uparrow;\ [\neg re];\ a0\downarrow]] \ ,$$

$$int1 \equiv *[[re \wedge x \longrightarrow a1\uparrow;\ [\neg re];\ a1\downarrow]] \ ,$$

$$
\begin{array}{lll}
SEL \equiv & *[[a0 & \longrightarrow r0\uparrow;\ [\neg a0 \wedge \neg a1];\ r0\downarrow \\
& \mid a1 & \longrightarrow r1\uparrow;\ [\neg a1 \wedge \neg a0];\ r1\downarrow \\
&]] \ . &
\end{array}
$$

The parallel composition of $int0$, $int1$, and SEL is not strictly equivalent to $sync1$ because, in the decomposition, the control can advance simultaneously to $a0\uparrow$ and $a1\uparrow$, which is not possible in the original $sync1$. Consequently, the arbitration takes place between $a0$ and $a1$ 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 $a0$ and $a1$ are false before resetting the output $r0$ or $r1$.

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:

$$
\begin{aligned}
r1\_ \wedge a0 &\rightarrow r0\_\downarrow \\
\neg r1\_ \vee (\neg a0 \wedge \neg a1) &\rightarrow r0\_\uparrow \\
r0\_ \wedge a1 &\rightarrow r1\_\downarrow \\
\neg r0\_ \vee (\neg a1 \wedge \neg a0) &\rightarrow r1\_\uparrow
\end{aligned}
$$

The complete circuit for the synchronizer comprising the two integrators, the bistable device and the metastability filter is shown in Figure 38. This is only one in a family of solutions[43]. A completely different solution is proposed in [48].

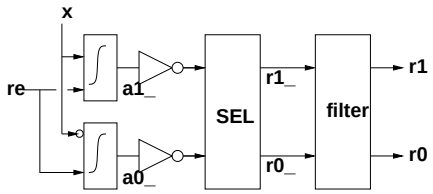

Figure 38: A synchronizer circuit consisting of two integrators (producing stable copies a1 and a0 of x and its inverse), a bistable device, and a filter.

9.8 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 Figure 39.

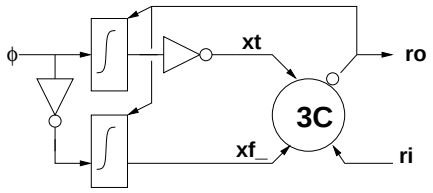

Figure 39: A conceptual implementation of the clock-handshake interface: The two integrators produce stable copies of the clock and its inverse; the 3-input C-element implements the four-phase handshake and produces the control input for the integrators.

The 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 Figure 40.

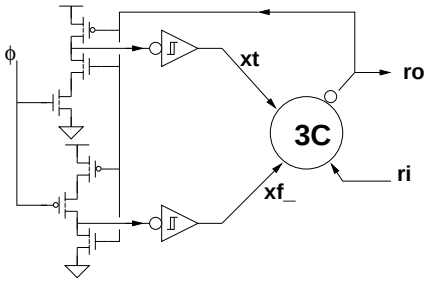

Figure 40: A practical implementation of the clock-handshake interface: The outputs of the integrators have been augmented with Schmitt-trigger inverters to guarantee quick transitions on xt and xf, and the inverter on the input of the bottom integrator has been eliminated.

10 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 [57, 58]; to this day, the common SCSI protocol for peripheral communications supports handshake-based asynchronous signalling [61]. 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 (a) there is no possibility of synchronization failure; or (b) 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.

10.0.1 Asynchronous-Synchronous Interface Using Two-Phase Non-Overlapping Clocks

Consider a synchronous system that operates off two-phase (ϕ_0 and ϕ_1 and their inverses) non-overlapping clocks. This is the standard method of designing custom CMOS circuits and is treated in detail by Mead and Conway [35] and Glasser and Dobberpuhl [65]. 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 non-overlap time between the phases to compensate for unexpected skews in the manufactured part.

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 1):

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

(We use the vertical thin bar $\mid$ 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 Figure 41. 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. Figure 42 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).

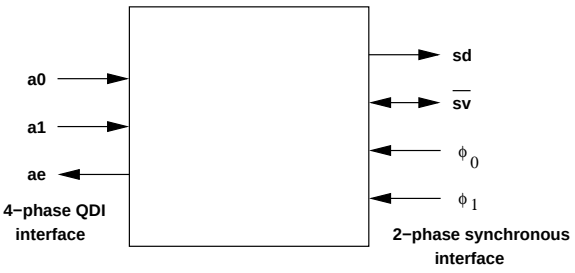

Figure 41: Block diagram of interface between four-phase QDI and two-phase synchronous pulse handshakes.

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 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 Figure 41, we can see that this wait can guarantee non-interference 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.

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

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 a 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 non-negated $a1$ 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 (Figure 43), 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 Figure 44. 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 Figure 43). 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).

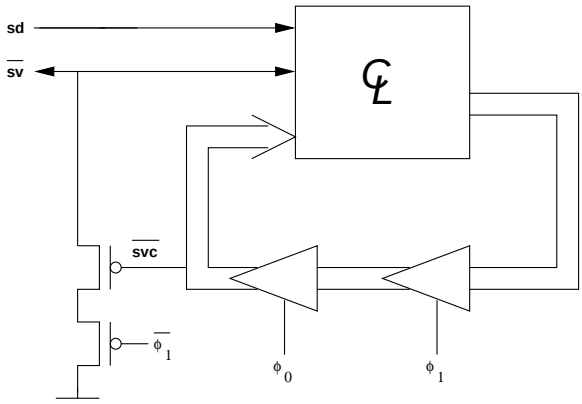

Figure 42: Connecting synchronous circuit with two-phase nonoverlapping clock to the asynchronous interface; logic between ϕ_1 and ϕ_0 latches not shown.

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. Secondly, 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 Figure 45. Finally, we should note that the designer who wishes to use domino logic between $sv\_$ and the next clock phase must proceed very carefully, as in our current design $sv\_$ can have a downwards glitch.

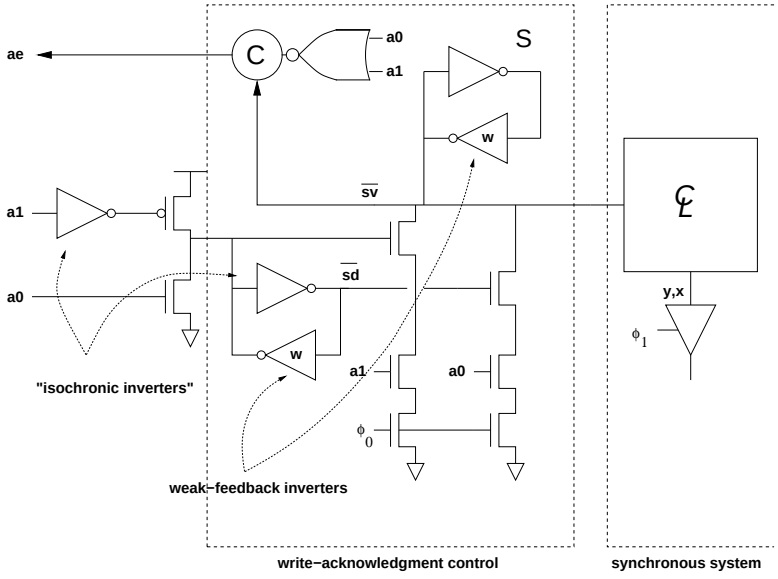

Figure 43: Detailed implementation of asynchronous-synchronous interface controller.

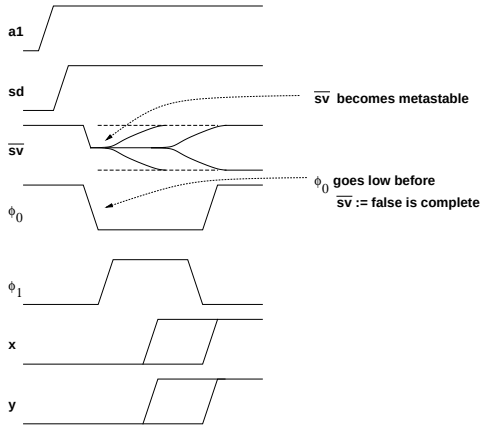

Figure 44: Timing diagram describing events leading up to synchronization failure.

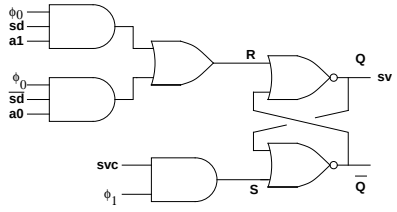

Figure 45: Use of S-R latch to improve metastability resolution time.

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.

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 ([53, 13, 66]...), this can be avoided.

10.1 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 Figure 46; 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.

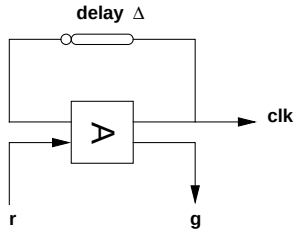

Figure 46: Safely stoppable clock oscillator.

10.1.1 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 Figure 47. We keep the design as similar as possible to the synchronizer-based described in Section 10.0.1. We assume a clock generator of the type described by Mead and Conway [35]. 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 10.0.1. 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 Figure 48; it is identical to the asynchronous controller for the synchronizer-based solution, except that the C-element driving *ae* has one additional input.

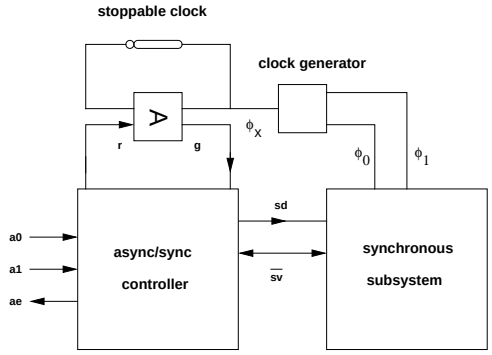

Figure 47: Block diagram of stoppable-clock interface for multiphase clocks.

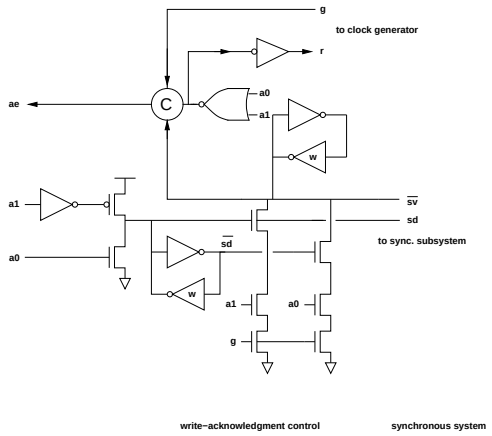

Figure 48: Circuit diagram of controller for stoppable-clock interface.

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.

10.1.2 Interfaces Using Edge-Triggered Methodology

In the previous section, we explored the design of asynchronous-synchronous interfaces using a clocking methodology with multiphase non-overlapping 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 non-overlapping clocks, each edge-triggered memory element, called an edge-triggered flip-flop, itself has a timing race. The timing races in edge-triggered logic give rise to timing constraints called *setup* and *hold* times, whereas non-overlapping 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 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.

10.2 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 Figure 45 with the multiplexed arbiter presented in Section 9.4, as shown in Figure 49, where $ARB1$ is an instance of the multiplexed arbiter, whereas $ARB0$ is the normal arbiter used in the stoppable-clock generator.

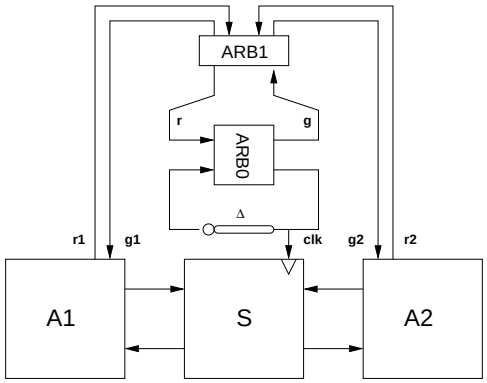

Figure 49: Sharing a stoppable clock from several interfaces; tree arbiter version.

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 (a) can occur only interspersed with synchronous cycles; and (b) are mutually exclusive.

A different implementation with multiple interfaces is shown in Figure 50. 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

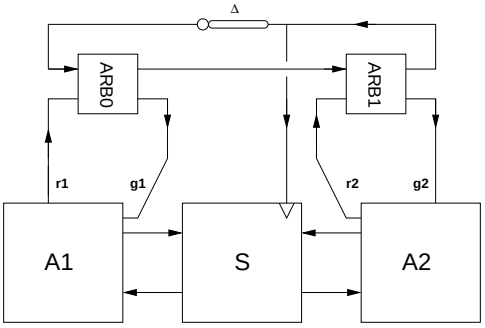

Figure 50: Sharing a stoppable clock from several interfaces; sequential arbitration version.

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 [39].

10.3 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[67]: 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, temperature...) 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 9.8 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 Figure 51(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 Figure 51(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 Figure 52. 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 (Figure 53). 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

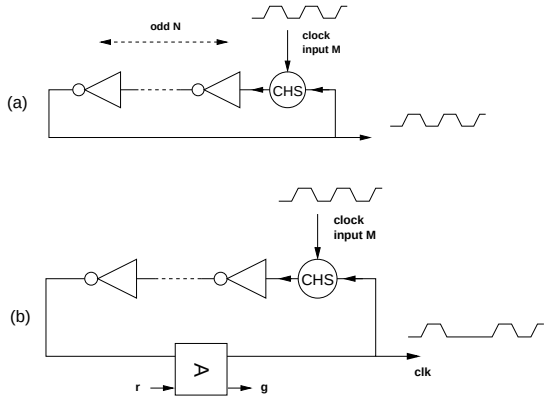

Figure 51: (a) Weakly synchronized "ring oscillator;" (b) Weakly synchronized stoppable-clock generator.

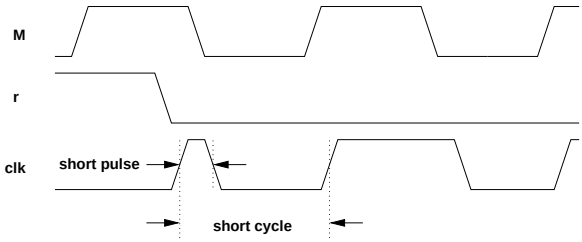

Figure 52: Timing diagram showing weakly synchronized stoppable-clock generator generating a very short clock pulse.

in the synchronous part).

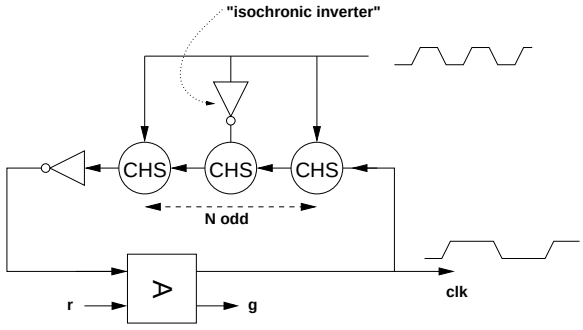

Figure 53: Improved weakly synchronized stoppable-clock generator.

10.4 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.

10.5 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 a SoC with multiple clock domains all driven off a single master (but with unknown inter-domain 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 [66]. 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 re-used 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 [53] described a design that takes advantage of this effect, using interleaved synchronizers.

11 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.

As was said earlier, 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 "asynchronous" 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.

12 Acknowledgment

The research in asynchronous VLSI at Caltech has been supported by the Defense Advanced Research Projects Agency (DARPA). It is currently monitored by the Air Force Office of Scientific Research. Acknowledgment is due to Jonathan Dama, Wonjin Jang, Mark Josephs, Mohsen Naderi, Karl Papadantonakis, and Piyush Prakash for their excellent comments on the paper.

References

[1] The 'Asynchronous' Bibliography, www.win.tue.nl/async-bib/, 2004.

[2] H. C. Brearley. "ILLIAC II: A short description and annotated bibliography," *IEEE Transactions on Computers,* 14(6):399-403, 1965.

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

[4] E. Brunvand and R. F. Sproull. "Translating concurrent programs into delay-insensitive circuits," *Proc. ICCAD,* IEEE Computer Society Press, 1989.

[5] S. M. Burns and A. J. Martin. "Syntax-directed translation of concurrent programs into self-timed circuits," *Advanced Research in VLSI* (J. Allen and F. Leighton, eds.) MIT Press, 1988.

[6] T. J. Chaney and C. E. Molnar. "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Transactions on Computers*, **C-22**(4):421–422, 1973.

[7] Daniel M. Chapiro. Globally-Asynchronous, Locally-Synchronous Systems. Ph.D. thesis, Stanford University, Stanford CS Technical Report STAN-CS-84-1026, 1984.

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

[9] Fu-Chiung Cheng. "Practical Design and Performance Evaluation of Completion Detection Circuits," *IEEE Int. Conf. on Computer Design (ICCD),* 1998.

[10] W.A. Clark. Macromodular computer systems. *AFIPS Conf. Proc: 1967 Spring Joint Computer Conf.,* vol 30, pp335-336. Academic Press, New York, 1967.

[11] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces.* Springer-Verlag, 2002.

[12] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev. "Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers." *IEICE Transactions on Information and Systems,* vol. E80-D,315-325, 1997.

[13] William J. Dally, John W. Poulton, *Digital Systems Engineering,* Cambridge University Press, 1998.

[14] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits,* vol. 56 of *CWI Tract,* 1989.

[15] Virantha Ekanyake *et al.* "BitSNAP: Dynamic Significance Compression for a Low-Energy Sensor Network Asynchronous Processor." *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* Los Alamitos, Calif.: IEEE Computer Society Press, 2005.

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

[17] S. B. Furber *et al.* AMULET2e: An Asynchronous Embedded Controller, *Proc. Async '97*, IEEE CS Press,290-299, 1997.

[18] Jim D. Garside. "Processors," Chapter 15 in Jens Sparsø and Steve Furber, eds. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Boston, Mass.: Kluwer Academic Publishers, 2001.

[19] H. v. Gageldonk *et al.* "An Asynchronous low-power 80c51 Microcntroller," *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,*96-107, Los Alamitos, Calif.: IEEE CS Press, 1999.

[20] Mark R. Greenstreet. "Real-Time Merging," *Proc. Fifth Int. Symp. on Advanced Research in Asynchronous Circuits and Systems: ASYNC99,* Barcelona, Spain, 19–21 April 1999. Los Alamitos, Calif.: IEEE CS Press, 1999.

[21] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* vol.21,666-677, 1978.

[22] D. A. Huffman, "The synthesis of sequential switching circuits," in *Sequential Machines: Selected Papers,* (E.F. Moore, ed.), Addison-Wesey, 1964.

[23] R. M. Keller. "Towards a theory of speed-independent modules." *IEEE Transactions on Computers*, C-23,21-33, 1974.

[24] Clinton Kelly IV, Virantha Ekanyake, Rajit Manohar. "SNAP: A Sensor Network Asynchronous Processor." *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* Los Alamitos, Calif.: IEEE CS Press, 2003.

[25] Andrew M. Lines. "Pipelined Asynchronous Circuits," MS Thesis, Caltech, 1997.

[26] Rajit Manohar and Alain J. Martin. "Quasi-delay-insensitive circuits are Turing-complete," *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* Los Alamitos, Calif.: IEEE CS Press, 1996.

[27] A. Marshall, B. Coates, and P. Siegel. "Designing An Asynchronous Communications Chip," *IEEE Design and Test of Computers*, **11**(2):8-21, 1994.

[28] Alain J. Martin. "The design of a self-timed circuit for distributed mutual exclusion," *Proc. of 1985 Chapel Hill Conferrence on VLSI,* (H. Fuchs, ed.), Computer Science Press, 1985.

[29] A. J. Martin. "Programming in VLSI: From communicating processes to self-timed VLSI circuits," in *Concurrent Programming,* (Proc. 1987 UT Year of Programming Institute on Concurrent Programming), C. A. R. Hoare, ed., Reading, Mass.: Addison-Wesley, 1989.

[30] A. J. Martin. "The limitations to delay-insensitivity in asynchronous circuits," in *Sixth MIT Conf. on Advanced Research in VLSI,* W. J. Dally, Ed. Cambridge, Mass.: MIT Press, 1990.

[31] Alain J. Martin *et al.* The design of an asynchronous microprocessor. In Charles L. Seitz, ed., *Advanced Research in VLSI: Proc. Decennial Caltech Conf. on VLSI*,351-373. Cambridge, Mass.: MIT Press, 1991.

[32] A. J. Martin *et al.* The Design of an Asynchronous MIPS R3000 Processor. *Proc. 17th Conf. on Advanced Research in VLSI.* Los Alamitos, Calif.: IEEE CS Press, 1997.

[33] Alain J. Martin. Synthesis Method for Self-timed VLSI Circuits. *Proc.1987 IEEE Int. Conf. on Computer Design (ICCD 87)*, 224-229, 1987.

[34] Alain J. Martin, Mika Nyström, Catherine G. Wong, "Three Generations of Asynchronous Microprocessors," *IEEE Design & Test of Computers, Special Issue on Clockless VLSI Design,* 2003.

[35] Carver Mead and Lynn Conway, *Introduction to VLSI Systems.* Reading, Mass.: Addison-Wesley, 1980.

[36] Raymond E. Miller, *Switching Theory,* vol.2, Chapter 10, Wiley, 1965.

[37] D.E.Muller and W.S.Bartky. A Theory of Asynchronous Circuits. *Proc. Int. Symp. on the Theory of Switching,* pages 24-243, Cambridge, MA, April 1959. Harvard University Press.

[38] Chris J. Myers. *Asynchronous Circuit Design.* Wiley-Interscience, 2001.

[39] J. Muttersbach, T. Villiger, W. Fichner, "Practical design of globally-asynchronous locally-synchronous systems." *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,*52-59, Los Alamitos, Calif.: IEEE CS Press, 2000.

[40] T. Nanya *et al.* TITAC: Design of a quasi-delay-insensitive microprocessor, *IEEE Design and Test of Computers,* IEEE, 1994.

[41] T. Nanya *et al.* TITAC-2: A 32-bit Scalable-Delay-Insensitive Microprocessor, *HOT Chips IX,*19-32, 1997.

[42] S. M. Nowick, M. B. Josephs, C. H. van Berkel, eds. "Special Issue: Asynchronous Circuits and Systems," *Proc. of the IEEE,* vol.16, 1999.

[43] Mika Nyström and Alain J. Martin."Crossing the synchronous-asynchronous divide," *Workshop on Complexity Effective Design,* 2002.

[44] Mika Nyström and Alain J. Martin. *Asynchronous Pulse Logic,* Kluwer, Boston, 2001.

[45] Piyush Prakash and Alain J. Martin. "Slack matching quasi-delay-insensitive circuits," to appear in: *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* 2006.

[46] Martin Rem. "Concurrent computations and VLSI circuits," *Control Flow and Data Flow: Concepts of Distributed Programming* (M. Broy, ed.), vol. F14 of *NATO ASI Series,*399-437, Springer-Verlag, 1985.

[47] M. Renaudin, P. Vivet, F. Robin, "ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor," *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,*22-31, Los Alamitos, Calif.: IEEE CS Press, 1998.

[48] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. "Q-Modules: Internally Clocked Delay-Insensitive Modules," *IEEE Transactions on Computers*, **37**(9):1005-1018, September 1988.

[49] S. Rotem *et al.* "RAPPID: An asynchronous instruction length decoder," *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* 60-70, Los Alamitos, Calif.: IEEE CS Press, 1999.

[50] Charles L. Seitz. "System timing," Chapter 7 in [35].

[51] J. L. A. v. d. Snepscheut. *Trace Theory and VLSI Design,* vol.200 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[52] Jens Sparsö and Steve Furber, eds. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Boston, Mass.: Kluwer, 2001.

[53] M. J. Stucki and J. J. Cox. "Synchronization strategies," *Proc. First Caltech Conf. on Very Large Scale Integration* (C.L.Seitz, ed.),375-393, 1979.

[54] I. E. Sutherland and S. Fairbanks. "GasP: A Mnimal FIFO Control,"*Proc. 10th Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,*, Los Alamitos, Calif.: IEEE CS Press, 2004.

[55] J. T. Udding. "A formal model for defining and classifying delay-insensitive circuits," *Distributed Computing,* vol.1, 4, pp. 197-204, 1986.

[56] T. E. Williams and M. A. Horowitz. "A zero-overhead self-timed 160ns 54b CMOS divider," *IEEE Journal of Solid-State Circuits,* Vol.26, 1991.

[57] Digital Equipment Corporation. *PDP-11 Peripherals and Interfacing Handbook,* DEC part number 112-01071-1854 D-09-25. Maynard, Mass.: Digital Equipment Corporation, 1971.

[58] Digital Equipment Corporation. *PDP-11 Bus Handbook,* DEC part number EB-17525-20/79 070-14-55. Maynard, Mass.: Digital Equipment Corporation, 1979.

[59] Arithmetic Processor 166 Instruction Manual, Digital Equipment Corporation, 1960.

[60] Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, 1969.

[61] Technical Committee T10, American National Standards Institute. Draft report X3T9.2/86-109, *Small Computer System Interface - 2.* American National Standards Institute, 1989.

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

[63] Larry R. Dennison, William J. Dally, and Thucydides Xanthopoulos: Low-latency plesiochronous data retiming. *Proc. Advanced Research in VLSI 1995:*304-315.

[64] Stephen A. Ward and Robert H. Halstead. *Computation Structures.* Cambridge, Mass.: MIT Press, 1989.

[65] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Reading, Mass.: Addison Wesley, 1985.

[66] Tiberiu Chelcea and Steven M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* **12**(8), August 2004,857-873.

[67] Frank K. Gürkaynak *et al.* "GALS at ETH Zurich: Success or Failure?" to appear in: *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* 2006.