Crossing the Synchronous-Asynchronous Divide

Mika Nyström, Alain J. Martin*

Abstract

The increasing complexity of modern VLSI systems has caused designers to reevaluate the age-old decision of using a single, central clock and instead turn to asynchronous or globally asynchronous, locally synchronous (GALS) designs. Communication between synchronous and asynchronous subsystems is difficult to do reliably and efficiently. In this paper, we review the background for the difficulty—metastability—and study new solutions to two different problems: (1) the synchronizer, which synchronizes a signal that can change at an arbitrary time into an asynchronous handshaking domain; and (2) an asynchronous clock-pulse generator.

1: Introduction

The most difficult aspect of computer design has always been *timing.* Many early computers (e.g., the ILLIAC and the DEC PDP-6) were asynchronous; these systems' designers felt that asynchronous machines were modular because timing issues could be localized to small parts of the machines. As late as the 1980s, asynchronous bus protocols were still common (such as DEC's UNIBUS and the original SCSI protocol, which is still supported by modern SCSI devices). But the difficulties of crossing from one clock domain to another spelled doom for asynchronous protocols, at least in small systems such as PCs, and today the standard approach is to provide the system designer with a hierarchy of clocks, all driven through "gearboxes" by a single central master. (The gearboxes are circuits that generate slave clocks, which are rational multiples of the master clock.) In larger systems, such as local-area networks, this is impossible, and such systems must cope with having several independent clock domains.

The trend in VLSI is that the chips get larger, the clocks get faster, and everything gets more complicated. We can expect that tomorrow's chips will look like today's local-area networks: the number of clock cycles required to get from one end of the die to the other will increase dramatically. This means that the gearbox approach will become more and more difficult to maintain, because a single wire might span several clock cycles, and the designer will find it difficult to ensure that, for instance, setup and hold times are maintained.

A way to deal with the increasing design complexity of VLSI systems is to bring back asynchrony. Several groups have had considerable success with entirely asynchronous systems [19, 4, 20, 5]; others have pursued the globally asynchronous, locally synchronous (GALS) paradigm, first suggested by Chapiro in 1984 [3]. In either case, the issue of interfacing synchronous and asynchronous domains arises.

*The authors are with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125, U.S.A.

Synchronous and asynchronous design methodologies are based on a simple principle: design composable subsystems in such a way that *if* the subsystems' environments satisfy certain assumptions, *then* the subsystems themselves will present such environments to their neighbors. For synchronous systems, these assumptions take the form of the circuits' having to maintain legal logic levels within certain setup and hold times. One might think that "asynchronous" circuits naturally make weaker demands on their environments since the clock has been removed. This is not necessarily so. In all asynchronous design methodologies, the synchronous level and timing requirements are replaced by certain handshaking requirements—requirements on the ordering of signal transitions. This means that sampling a signal from a synchronous system within the asynchronous framework is fraught with difficulty, much like the converse operation of sampling an asynchronous signal within the synchronous framework. With care, however, an asynchronous system can sample an unsynchronized signal with zero probability of synchronization failure.

We shall review the main source of trouble: metastability; secondly, we shall investigate efficient solutions to a few typical problems: (1) the synchronizer problem, which is concerned with "absorbing" an arbitrarily varying signal into a four-phase signalling scheme; and (2) implementation of an asynchronous timer without introducing metastability.

2: Metastability

In the late 1960's, designers of synchronous systems that engaged in high-speed communications between independent clock domains found a new class of problems related to accepting an unsynchronized signal into a clock domain. A device that can reliably and with bounded delay order two events in time cannot be constructed under the assumptions of classical physics. The basic reason for this is that such a device would have to make a discrete decision—which event happened first—based on a continuous-valued input—the time. Given an input that may change asynchronously, if we attempt to design a device that samples its value and returns it as a digital signal, we must accept that the device either may take unbounded time to make its decision or that it may sometimes produce values that are not legal ones or zeros but rather something in between. The failure of such a device to produce a legal logic value is called *synchronization failure*; Chaney and Molnar provided the first convincing experimental demonstration of synchronization failure in 1973 [2]. Synchronous designers must accept a certain risk of system failure, which can be traded against performance, as discussed in the literature [21].

Synchronization failure may be avoided by making the sampling system completely asynchronous. In such a system, no clock demands that the system make its decision after a certain, fixed amount of time and system operation can be suspended until the decision has been resolved. The device that determines whether a particular event happened before or after another is called an *arbiter*. A typical CMOS arbiter is shown in Figure 1. This is the familiar R-S latch with a filtering circuit on the output. In contrast to how this device is used in synchronous circuits, the arbiter is allowed to go into the metastable state if the two inputs arrive nearly simultaneously. The filtering circuit on the output (a pass-gate-transformed pair of NOR gate/inverter hybrids) ensures that the arbiter outputs u and v do not change until the internal-node voltages (on s and t) are separated by at least a p-transistor threshold voltage—which means that the internal nodes have left the metastable state. At that time, the arbiter has "made up its mind," and there is no possibility of an output glitch.

If the rest of the system can wait until the arbiter asserts one of its outputs, which could take forever, then there is no possibility of synchronization failure. We stress that even though the arbiter *could* take forever, in practice, it rarely takes very long to exit the metastable state. In fact, this is

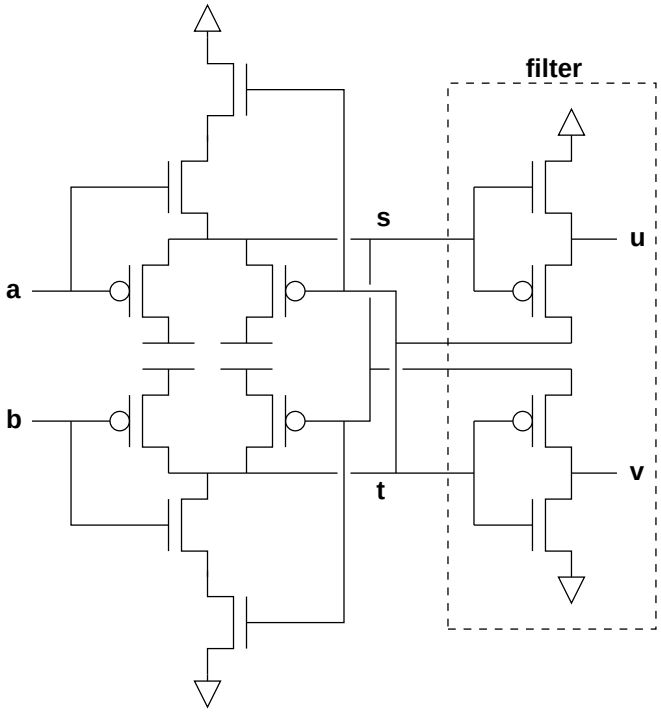

Figure 1. CMOS arbiter circuit.

the reason that asynchronous implementation of systems that require arbitration is attractive—the *average* delay of the arbiter is likely to be much smaller than the latency that would be required to reduce the probability of synchronization failure in a synchronous implementation to acceptable levels.

The proper operation of the arbiter circuit depends on the fact that the inputs are stable, i.e., that the inputs remain asserted until the arbiter has acknowledged the input by making its decision. If one of the requests is withdrawn before the arbiter has made its decision, the arbiter may fail (one or both of the outputs may glitch). Figure 2 shows an example of what happens if an unstable input is sampled with a normal arbiter. In this figure, Go represents the input to the arbiter, a 100 ps pulse, $x.r1\_$ represents the switching internal node (between the R-S latch and the filter stage), and $x.x1$ represents the output, which glitches. (The simulation parameters used here are for HP's 0.6-μm CMOS process.)

Expressed as a Production Rule Set (PRS)[18], the CMOS arbiter may be written

$$\begin{aligned} a \wedge t &\mapsto s\downarrow \\ b \wedge s &\mapsto t\downarrow \\ \neg a \vee \neg t &\mapsto s\uparrow \\ \neg b \vee \neg s &\mapsto t\uparrow \quad . \end{aligned}$$

A rule $G \mapsto s\downarrow$ means that the variable s is set to **false** when the condition G is **true**. Rules of the form $G \mapsto s\downarrow$ correspond to pull-down chains, and $G \mapsto s\uparrow$ correspond to pull-up chains. These production rules correspond to the circuit shown in Figure 1.

The arbiter is specified by the following *handshaking expansion* (HSE)[18]:

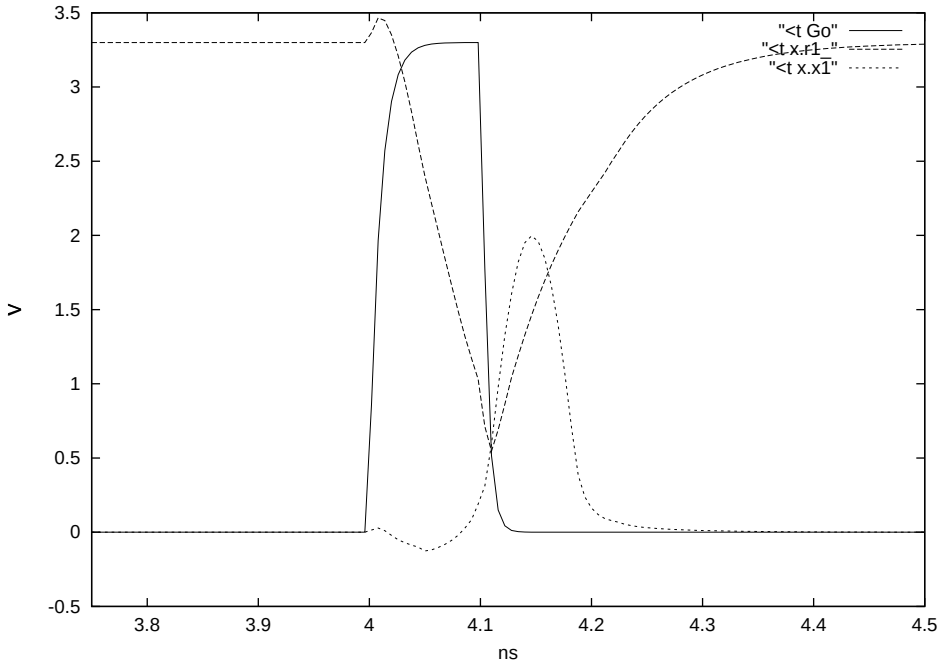

Figure 2. Waveforms of a misbehaving arbiter.

$$
\begin{array}{l}
*[[a \longrightarrow u\uparrow; [\neg a]; u\downarrow \\
\quad | b \longrightarrow v\uparrow; [\neg b]; v\downarrow \\
]]
\end{array}
$$

2.1 The Synchronizer

As we have seen, the arbiter can misbehave (have glitches) when an input is asserted and then withdrawn before being acknowledged. This means that we cannot use an arbiter to sample a completely unsynchronized external signal, something that is for instance required by the interrupt mechanism on a MIPS microprocessor [9]. A circuit that solves this more difficult problem is called a *synchronizer.*

The synchronizer has two inputs: a control input and an input signal that is to be sampled. The specification of the synchronizer is, informally, that it waits for the control signal to be asserted and then samples the input. We are attempting to build the circuit so that it can be part of a QDI asynchronous system; therefore, the synchronizer produces a dual-rail output with the two rails representing the value of the sample input: either $\mathtt{true}$ or $\mathbf{false}$.

Using handshaking expansions, the program for the synchronizer is given by

$$
\begin{array}{l}
*[[re \wedge \neg x \longrightarrow r0\uparrow; [\neg re]; r0\downarrow \\
\quad | re \wedge x \longrightarrow r1\uparrow; [\neg re]; r1\downarrow \\
]]
\end{array}
$$

where x is the input being sampled, re is the control input, and the pair $(r0, r1)$ is the dual-rail output. When the environment asserts re, the synchronizer springs into action and samples x, returning the observed value by asserting either $r0$ or $r1$. What makes the implementation of this circuit challenging is that x may change from **true** to **false** and vice versa at any time. If the input

has a stable **true** value within a finite, bounded interval around the time the control input arrives, the circuit asserts $r1$; if the input is a stable **false**, the circuit asserts $r0$; otherwise, the circuit asserts either $r0$ or $r1$, but not both. In practice, the "confusion interval" will be a very short time indeed, approximately the delay of the single inverter used to invert the input. The confusion interval is analogous to the setup and hold times of a latch; however, as opposed to a latch, the synchronizer is required to operate correctly (albeit non-deterministically) even if the input changes in this interval.

What makes the implementation of the synchronizer difficult is that it must work correctly when the input changes during the confusion interval. Implementing a synchronizer properly is so difficult that it is usually avoided: for instance, Marshall *et al.* avoid it by detecting the withdrawal of a request and resetting their entire system [13].

3: Top-down derivation of a synchronizer

By keeping in mind the metastability argument of the previous section, we arrive at a correct synchronizer design through a top-down derivation.

The synchronizer is given by the following handshaking expansion:

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

whose environment is described by

$$ENV \equiv *[re\uparrow; [r0 \vee r1]; re\downarrow; [\neg r0 \wedge \neg r1]] \ .$$

Unfortunately, this specification of $SYNC$ is not directly implementable. To see why, consider that the program has to do two things to advance from the wait for $re \wedge x$ to $r0\uparrow$: first, it must "lock out" the second guard, so that $r1\uparrow$ cannot happen; secondly, it must actually perform the assignment $r0\uparrow$. If x should change after the second guard has been locked out but before the first guard has proceeded, the program will deadlock. There is an inherent race condition that we have to remove; it is obvious that we shall have to remove it by introducing intermediate states before $r0\uparrow$ and $r1\uparrow$.

We introduce explicit signals $a0$ and $a1$ that are used to hold the values $re \wedge \neg x$ and $re \wedge x$ respectively. We augment $SYNC$ with assignments to $a0$ and $a1$:

$$\begin{aligned} SYNC1 \equiv & \\ & *[[re \wedge \neg x \longrightarrow a0\uparrow; [a0]; r0\uparrow; [\neg re]; a0\downarrow; [\neg a0]; r0\downarrow \\ & \ | re \wedge x \longrightarrow a1\uparrow; [a1]; r1\uparrow; [\neg re]; a1\downarrow; [\neg a1]; r1\downarrow \\ & \ ]] \end{aligned}$$

We introduce an explicit handshaking expansion to use the newly introduced signals $a0$ and $a1$ to produce outputs $r0$ and $r1$. The result is:

$$\begin{aligned} SYNC2 \equiv *[[& re \wedge \neg x \longrightarrow a0\uparrow; [\neg re]; a0\downarrow \\ & | re \wedge x \longrightarrow a1\uparrow; [\neg re]; a1\downarrow \\ &]] \end{aligned}$$

$$\begin{aligned} SEL \equiv *[[& a0 \longrightarrow r0\uparrow; [\neg a0]; r0\downarrow \\ & [] a1 \longrightarrow r1\uparrow; [\neg a1]; r1\downarrow \\ &]] \end{aligned}$$

$$SYNC1 \equiv \ SYNC2 \ \| \ SEL$$

(The bar "[]" means that both $a0$ and $a1$ cannot be high at the same time when SEL is executing the selection statement.) We should like to arbitrate between $a0$ and $a1$ instead of between $re \wedge \neg x$ and $re \wedge x$, and not implement $SYNC2$ as written. Instead of $SYNC2$, we use the following production rules that permit the different parts of $SYNC2$ to execute concurrently. Specifying the circuit behavior as a handshaking expansion is cumbersome, so we proceed directly to the production rules:

$$\begin{aligned} re \wedge \neg x &\mapsto a0\uparrow \\ \neg re &\mapsto a0\downarrow \end{aligned}$$

$$\begin{aligned} re \wedge x &\mapsto a1\uparrow \\ \neg re &\mapsto a1\downarrow \end{aligned}$$

To make the rules CMOS-implementable, we introduce an inverter to generate $x\_$ in the first production rule. Given that these rules all execute concurrently, we examine the behavior of SEL in greater detail.

Signals $a0$ and $a1$ are independent from each other, in the sense that they can be both true at the same time if x is sampled during the confusion interval. Also, because x can be sampled during a transition, the transitions $a0\uparrow$ and $a1\uparrow$ are not always completed, but we assume that at least one of them completes eventually. The essential fact about the signals $a0$ and $a1$ is that they are monotonically increasing as long as re is **true**. In fact, $a0$ can be thought of as the "integral" of $\neg x \wedge re$: as long as $\neg x$ holds, $a0$ increases; should x begin to hold instead, $a1$ will increase instead.

SEL is similar to but not identical to an arbiter. In an arbiter, when both inputs are high, the arbiter selects both inputs one after the other in arbitrary order since a request is never withdrawn (see Section I). In SEL, on the other hand, when both inputs are high, *only one* should be selected. Hence, we must check that both $a0$ and $a1$ are **false** before resetting the output of SEL. The new process is:

$$\begin{aligned} SEL \equiv *[[& a0 \longrightarrow r0\uparrow; [\neg a0 \wedge \neg a1]; r0\downarrow \\ | & a1 \longrightarrow r1\uparrow; [\neg a1 \wedge \neg a0]; r1\downarrow \\ ]] & \end{aligned}$$

(Note that we have re-introduced the "|" indicating that the selection between $a0$ and $a1$ is nondeterministic.) SEL can be implemented directly as a bistable device, and the production rules are:

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

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

The circuit corresponding to the production rule set for the synchronizer is shown in Figure 3, where we have added the filter stage necessary to block the metastable state from reaching the digital outside world.

3.1 Summary

Let us summarize what we have done. We started with an input, x, which we assumed could change at any time—in fact, x need not even have a defined logic value. The problem we set out to solve was to sample this x; by this we mean that if x holds a stable legal logic value, then our

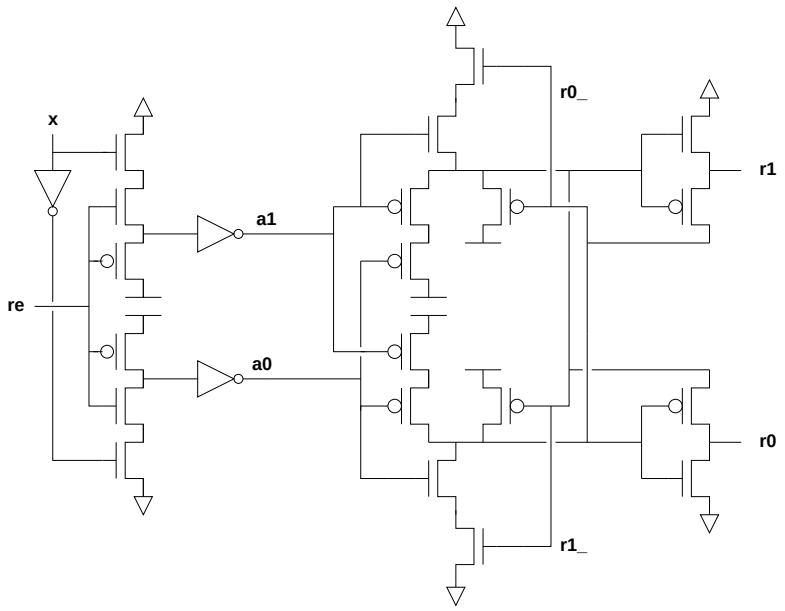

Figure 3. Synchronizer

circuit returns that value; if x does not hold a stable legal logic value, we do not care whether our circuit returns zero or one, as long as it returns one of the two rather than failing in some disastrous way. We accomplished this by "integrating" the signal x into the two monotonic intermediaries $a0$ and $a1$; being monotonic, we could apply straightforward techniques to them.

In a separate publication [23], we review different versions of this circuit (the one we have seen here is only the most basic) and also contrast the various versions to the earlier design by Rosenberger *et al.* [24]. Rosenberger *et al.*'s design is a sense-amplifier design, which draws static power and depends on transistor-ratioing assumptions that make it difficult to build a reliable circuit realization [10]. Ours, on the other hand, is a normal pseudo-static CMOS circuit that draws no static power and poses no especially difficult analog design problems. We should remember, however, that Rosenberger's design is the only prior work that actually solves the problem of synchronizing a completely asynchronous signal with zero probability of failure; other designers have simply given up and, like their synchronous friends, accepted a nonzero probability of failure [13]. As we have already discussed, the zero probability of failure is not a merely academic property; rather, it allows the designer to increase the performance of the design by taking advantage of average-case performance rather than introducing extra latency to deal with once-in-a-lifetime metastability events.

We have seen that what makes building a synchronizer different from building an arbiter is that the synchronizer has to contend with requests that are withdrawn before they are granted. In the special case where we know that the withdrawal will happen while another request is being serviced, the full functionality of the synchronizer is not necessary [11, 6]. A synchronizer similar to the one described here was used to sample the interrupt inputs in the MiniMIPS processor [20].

4: An Asynchronous Clock Circuit

We shall study the specific case of interfacing a QDI asynchronous system with a synchronous environment. Specifically, we shall develop and analyze a simple circuit that allows a QDI asynchronous system to sample an external *clock signal.* Why should we want a QDI asynchronous system to do such a thing? The main reason is so that we can build accurate timers; for instance, the Intel 8051 microcontroller is specified to have timers that count incoming clock ticks and raise an interrupt when the count reaches a certain value [1].

4.1 Prior solutions

We are studying the problem of sampling an external, free-running clock signal with an asynchronous circuit. This problem can be solved using a synchronizer: we simply set up the synchronizer to sample the input repeatedly and count the number of times that we see a "1" followed by a "0" or vice versa.

Two problems are immediately evident:

1. The circuit can miss input transitions; if R is not read often enough, it is possible for x to transition from, e.g., **true** to **false** and back to **true** again.
2. If we wish to mitigate the first disappointment, we shall have to make our circuit read R very often. If it then turns out that, contrary to our original fear of x's changing too often, x actually changes very infrequently, then we shall be wasting a great deal of energy by inspecting x too frequently. (The sampling rate for the synchronizer must be higher than the maximum rate at which x can change.)

There is no completely satisfactory solution to the first problem: if the asynchronous system that is interested in the value of x cannot keep up with the changes on x, then transitions on x shall simply have to be missed.

From before, we know that the synchronizer is a subtle circuit that among other things includes a metastable element. The presence of the metastable element is required because the specification requires the synchronizer to sequence two events whose occurrence times may be arbitrarily close to each other. The circuit that we shall discuss now borrows parts from our earlier synchronizer design while avoiding the metastability.

4.2 A Synchronous & Asynchronous Event Generator

We shall consider designing a circuit that takes a synchronous clock signal and converts that into a stream of asynchronous events or handshakes. Using the Handshaking Expansion language (HSE), we write

$SYNCASYNC \equiv$

$*[\ [x];\ [\neg x];\ R\ ]$

where x is the synchronous clock input, and R denotes a handshake communication with the asynchronous environment. Obviously, if R takes too long to complete (so that x has time to go from **false** to **true** back to **false**), the circuit will miss a clock cycle, and it cannot proceed until the next cycle.

4.3 Input integrators

The synchronizer we have already studied includes the following element as its "first stage":

$INTEGRATE \equiv$

$*[[x \ \wedge \ \neg rt]; \ qt\uparrow; \ [rt]; \ qt\downarrow]$

$INTEGRATE$ waits for x to become **true** and then completes a handshake. $INTEGRATE$ may make progress even if x is not **true** long enough to fire $qt\uparrow$; if x is oscillating very rapidly, an implementation of $INTEGRATE$ may integrate x until it has accumulated enough charge to fire $qt\uparrow$.

As in the synchronizer, we further construct the following:

$NOTINTEGRATE \equiv$

$*[[\neg x \ \wedge \ \neg rf]; \ qf\uparrow; \ [rf]; \ qf\downarrow]$

In the synchronizer, $NOTINTEGRATE$ is realized with an instance of $INTEGRATE$ with an inverter on its input (we shall see that we can even avoid that here).

4.4 Control process

If we use $INTEGRATE$ and $NOTINTEGRATE$ to implement $[x]$ and $[\neg x]$, then we only need to add a control process to sequence the actions properly, and we say that $SYNCASYNC = INTEGRATE \parallel NOTINTEGRATE \parallel CONTROL$. The obvious implementation of $CONTROL$ is the following:

$CONTROL \equiv$

$$
\begin{array}{l}
*[[qt]; \ rt\uparrow; \ [\neg qt]; \ rt\downarrow; \\
\quad [qf]; \ rf\uparrow; \ [\neg qf]; \ rf\downarrow; \\
\quad ro\uparrow; \ [ri]; \ ro\downarrow; \ [\neg ri]]
\end{array}
$$

Here we have replaced the communication R with its implementation $ro\uparrow; [ri]; ro\downarrow; [\neg ri]$.

$CONTROL$ is a good candidate for reshuffling: let us bring out $[qt]; rt\uparrow$ from the first iteration and intertwine the t and f communications:

$CONTROL \equiv$

$$
\begin{array}{l}
[qt]; \ rt\uparrow; \\
\quad *[[\neg qt \wedge qf]; \ rt\downarrow, rf\uparrow; \\
\quad\quad [qt \wedge \neg qf]; \ rt\uparrow, rf\downarrow; \\
\quad\quad ro\uparrow; \ [ri]; \ ro\downarrow; \ [\neg ri]]
\end{array}
$$

We can continue the same process with the R communication:

$CONTROL \equiv$

$$
\begin{array}{l}
[qt]; \ rt\uparrow; \\
\quad *[[\neg qt \wedge qf \wedge \neg ri]; \ rt\downarrow, rf\uparrow, ro\uparrow; \\
\quad\quad [qt \wedge \neg qf \wedge ri]; \ rt\uparrow, rf\downarrow, ro\downarrow \ ]
\end{array}
$$

The "prolog" $[qt]; rt\uparrow$ disturbs the symmetry of the program; by removing it, we shall only find that we ignore the first rising edge of the input clock. We hence get the following program:

$CONTROL \equiv$

$$*[[\neg qt \wedge qf \wedge \neg ri]; \quad rt\downarrow, rf\uparrow, ro\uparrow;$$
$$\quad [qt \wedge \neg qf \wedge ri]; \quad rt\uparrow, rf\downarrow, ro\downarrow \ ]$$

Finally, it seems obvious that this program could have a particularly simple implementation if the signal senses are adjusted. The following works best:

$CONTROL \equiv$

$$*[[\neg qt \wedge \neg qf\_ \wedge \neg ri]; \quad rt\_\uparrow, rf\uparrow, ro\uparrow;$$
$$\quad [qt \wedge qf\_ \wedge ri]; \quad rt\_\downarrow, rf\downarrow, ro\downarrow \ ]$$

In other words, $CONTROL$ is simply an inverting three-input C-element; with this implementation, $rt\_$, rf, and ro become three names for the same circuit node.

4.5 Transistor-level implementation

Now let us clarify the choice of signal senses. We implement $INTEGRATE$ with the following production-rule set (PRS):

$$\begin{aligned} x \wedge rt\_ &\mapsto qt\_\downarrow \\ \neg rt\_ &\mapsto qt\_\uparrow \\ qt\_ &\mapsto qt\downarrow \\ \neg qt\_ &\mapsto qt\uparrow \end{aligned}$$

Similarly, $NOTINTEGRATE$ becomes the following:

$$\begin{aligned} \neg x \wedge \neg rf &\mapsto qf\downarrow \\ rf &\mapsto qf\downarrow \\ qf &\mapsto qf\_\downarrow \\ \neg qf &\mapsto qf\_\uparrow \end{aligned}$$

By inverting the sense of $NOTINTEGRATE$ relative to $INTEGRATE$, we avoid having to put an inverter on the input to $NOTINTEGRATE$; furthermore, since all the production rules are inverting, the circuit is directly implementable in CMOS circuits. Since the nodes $qt\_$ and qf are state-holding, we must add staticizers (bleeders) to them.

$CONTROL$ is implemented as the following PRS:

$$\begin{aligned} \neg qt \wedge \neg qf\_ \wedge \neg ri &\mapsto ro\uparrow \\ qt \wedge qf\_ \wedge ri &\mapsto ro\downarrow \end{aligned}$$

It is more convenient to make ri inverted; we should normally call this re (for "r enable"—when re is **true**, the circuit is *enabled* to produce an output).

4.6 Analog difficulties

It is not immediately obvious that $INTEGRATE$ and $NOTINTEGRATE$ will work correctly for any input waveform on x. The PRS implementation guarantees that x can only cause $qt\_$ to decrease and can only cause qf to increase; but is this enough for the circuit to be correct? This question can only be answered relative to the destination of $qt\_$ and qf. Unfortunately, this destination is a C-element: a noise-sensitive dynamic circuit. A very slowly decreasing $qt\_$—hence, a slowly rising qt—could, for instance, trigger the C-element well before qt reaches the Vdd rail. In this case, we should have to depend on a timing assumption to guarantee that qt gets to the Vdd

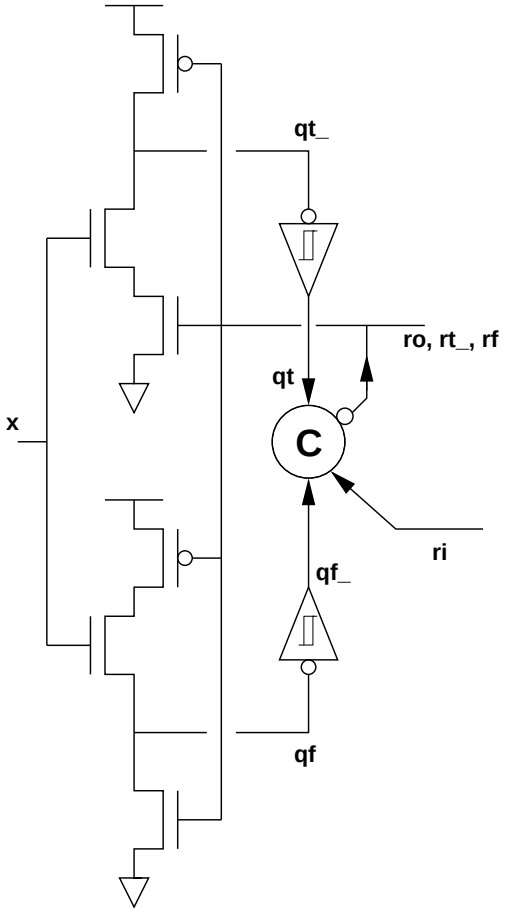

Figure 4. Asynchronous clock circuit.

rail before $qf\_$ switches; if it did not, the intermediate voltage could be interpreted as **false** by the C-element later.

We can remove the analog problem by implementing the two inverters

$$\begin{aligned} qt\_ &\mapsto qt{\downarrow} \\ \neg qt\_ &\mapsto qt{\uparrow} \\ qf &\mapsto qf\_{\downarrow} \\ \neg qf &\mapsto qf\_{\uparrow} \end{aligned}$$

with Schmitt triggers. With this change, we are guaranteed that qt and $qf\_$ switch quickly enough that neither node will be observed as both **true** and **false**. This is because of the following property of a Schmitt trigger: if the input changes *monotonically* from one legal logic value to the other, then the output will switch *quickly* between the logic values. However, a Schmitt trigger is not a magical device that can turn an arbitrary input waveform into a sequence of transitions between legal logic values—the property we are using only holds if the inputs are monotonic. This is why we have been careful to design $INTEGRATE$ and $NOTINTEGRATE$ so that their outputs are monotonic, regardless of their inputs. The transistor diagram of the final circuit is shown in Figure 4.

We should note that the analog problem we describe here is unlikely to cause practical difficul-

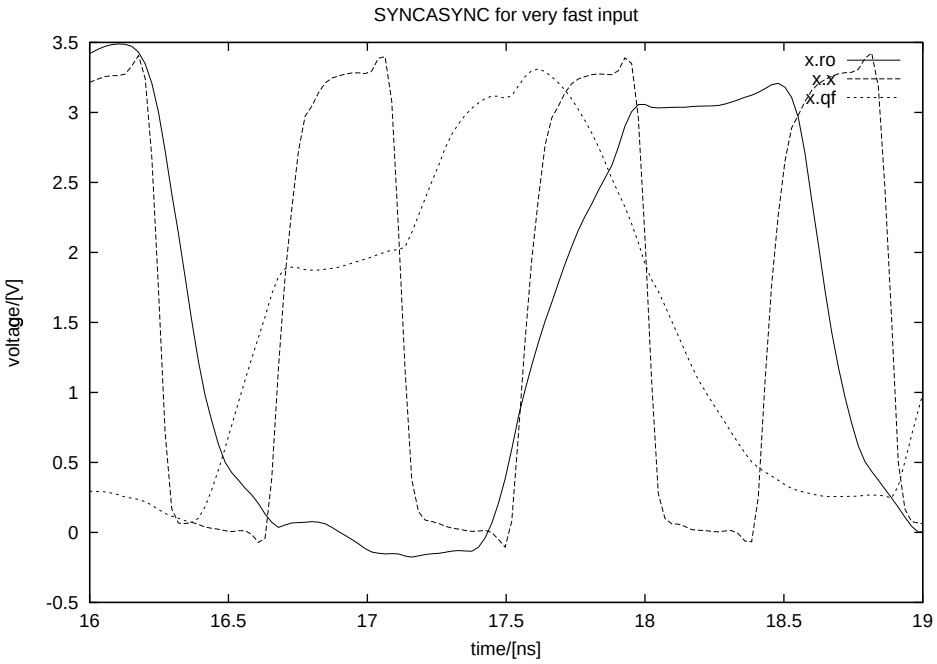

Figure 5. Very fast input signal applied to *SYNCASYNC.*

ties; only a very unfortunate selection of transistor sizes could allow qt to linger in an intermediate voltage range for an appreciable length of time. However, once we have substituted the Schmitt triggers we can be sure that the circuit is correct without recourse to anything beyond a simple monotonicity argument for $qt\_$ and qf and our well-tested QDI theory for the rest. (The synchronizer can also be improved, at least in theory, by similarly adding Schmitt triggers to it [23].)

An example of how the circuit handles a particularly difficult input is shown in Figure 5. This is the SPICE output of a simulation of the circuit using 0.6-μm parameters (for HP's CMOS14B process via MOSIS). A 1140 MHz input signal generated by a seven-stage ring oscillator is applied at node x; this is far faster than the maximum of about 400 MHz the circuit can handle in this technology (without any detailed transistor-size optimization). The nodes ro, x, and qf are shown in the graph. We see how qf begins to rise (because x is **false**); when qf has reached about halfway, x becomes **true** and qf's rising stops; finally, x becomes **false** again, and qf goes **true**. Since we have designed the circuit to generate a communication on R every time it detects a downward edge on x, it is qf's going **true** that eventually causes the upward transition on ro.

5: The Gray-Code Counter

The curious reader will wonder whether one can devise a way for an asynchronous system to read a free-running counter (e.g., a time-of-day register driven by an accurate oscillator) (a) with zero probability of synchronization failure, (b) without interrupting the counter, and (c) with an absolute guarantee that the time that was read was indeed contained in the register in a bounded neighborhood of the time that the request was made.

This particular problem can be solved with a synchronous Gray-code counter and either an array of synchronizers (one per bit of the counter) or a single synchronizer that is used once per bit of

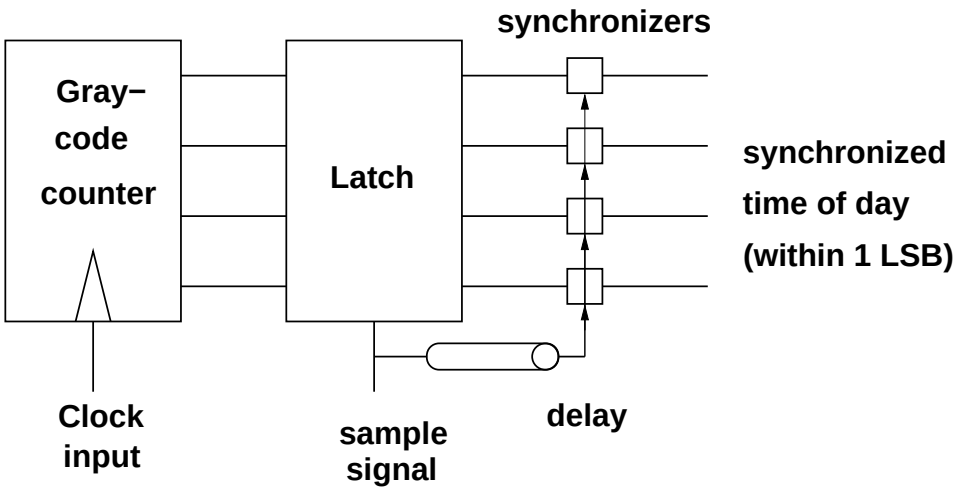

Figure 6. Gray-code scheme for reading time of day asynchronously.

the counter. The counter operates synchronously to its clock and cannot be interrupted. When the asynchronous system desires to know the time of day, it latches the current value of the counter. The latching results in at most one bit's being in an intermediate state between a valid logic **false** and a valid **true**. It is possible to detect that all the latched-in values have decayed, as they do in a latch, to valid logic values; at this time, the asynchronous system knows the time of day at the time of the request to within 1 LSB. (It is actually possible to omit the intermediate latch and simply use the synchronizer directly.) The scheme is illustrated in Figure 6.

Let us compare the Gray-code solution with the event-generator solution we have presented in this paper. The Gray-code solution requires a bundled-data timing assumption when we read the bits: we must assume that all the bits of the counter are read in simultaneously, or at least so quickly that at most two values are seen in the latches and synchronizers; the Gray-code solution furthermore has a rather complicated interface between the synchronous and asynchrous parts (which we should like to avoid if possible); finally, it requires us to compute the specific piece of information we desire on the "synchronous side" of the system before turning it over to the "asynchronous side" of the system. In contrast, the event-generator solution has a very simple synchronous-asynchronous interface: every clock pulse turns into an asynchronous event; everything we desire to compute we can compute on the asynchronous side of the system; but for this convenience we pay a finite probability of losing clock pulses (we can make this finite probability arbitrarily small by adding asynchronous buffering downstream of the event generator).

The remarkable thing about the asynchronous clock circuit presented in this paper is that it is nondeterministic—i.e., we cannot tell if an input transition will lead to an output action without reference to the actual timing that obtains in the given situation—yet it contains no arbiter, synchronizer, or other metastable device! The most closely related work to ours is Greenstreet's "real-time merging" circuit, which uses Schmitt triggers in a similar way but for a different purpose, namely nondeterministic merging of two data streams without metastability [7].

6: Conclusion

In this paper, we have studied two similar problems that involve accepting signals that change at arbitrary times into an asynchronous handshaking framework. The first design problem, the synchronizer, is one of the most difficult circuit-design problems in asynchronous design: a synchronizer is in some sense the most powerful circuit element imaginable since it allows the reliable reading of an input variable while placing *no* constraints on the timing behavior of that variable. The second design problem, the asynchronous clock circuit, is similar to the synchronizer in that the input can change at arbitrary times; however it is very different in the format of the output: instead of reading the input at a certain times, this circuit is reactive and produces an output when the input changes. This circuit solves the problem of building a timekeeper in the asynchronous framework very elegantly: it is simpler and easier to verify correct than the synchronous Gray-code counter, and it can be far more efficient than a synchronizer-based solution (whether synchronous or asynchronous) since its power consumption is proportional to the average rate of input transitions rather than to the maximum rate of input transitions.

Acknowledgments

The research described in this paper was supported in part by the Defense Advanced Research Projects Agency (DARPA) and monitored by the Air Force Office of Scientific Research. Rajit Manohar has contributed to the design of the synchronizer. The need for an "asynchronous clock circuit" was suggested by Karl Papadantonakis.

References

[1] *80C51 family programmer's guide and instruction set.* Philips Semiconductors, 1997.

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

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

[4] U. V. Cummings, A. M. Lines, and A. J. Martin. An Asynchronous Pipelined Lattice Structured Filter. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 1994.

[5] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design,* August 1993.

[6] 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.

[7] Mark R. Greenstreet. "Real-Time Merging," *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems: ASYNC99,* Barcelona, Spain, 19–21 April 1999. Los Alamitos, Calif.: IEEE Computer Society Press, 1999.

[8] S. Hauck. "Asynchronous Design Methodologies: An Overview," *Proceedings of the IEEE,* **83**(1):69–93, 1995.

[9] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[10] A. M. Lines. Personal communication, 2001.

[11] Rajit Manohar, Mika Nyström, and Alain J. Martin. Precise Exceptions in Asynchronous Processors. Erik Brunvand and Chris Myers, eds., *Proceedings of the 2001 Conference on Advanced Research in VLSI: ARVLSI 2001*, Salt Lake City, Utah, March 14–16, 2001. Los Alamitos, Calif.: IEEE Computer Society Press, 2001.

[12] Leonard R. Marino. "General Theory of Metastable Operation," *IEEE Transactions on Computers*, **C-30**(2):107–115, February 1981.

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

[14] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

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

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

[17] A. J. Martin. "Synthesis of Asynchronous VLSI Circuits," in *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.

[18] A. J. Martin. "Synthesis of Asynchronous VLSI Circuits," Caltech CS Technical Report Caltech-CS-TR-93-28. California Institute of Technology, 1993.

[19] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, ed., *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373. Cambridge, Mass.: MIT Press, 1991.

[20] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee. The Design of an Asynchronous MIPS R3000 Processor. *Proceedings of the 17th Conference on Advanced Research in VLSI*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

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

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

[23] Mika Nyström, Rajit Manohar, and Alain J. Martin. A Failure-Free Synchronizer. In preparation, 2002.

[24] 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.

[25] Charles L. Seitz. "System timing," chapter 7 in [21].