A Failure-Free Synchronizer

Mika Nyström, Rajit Manohar, Alain J. Martin

Abstract— **A synchronizer is a circuit that can sample an arbitrary unstable input signal. We present the first synchronizer without transistor-strength ratioing assumptions that is shown to allow zero probability of system failure when used in a fully asynchronous system. We present two solutions to the problem, differing in the degree of detailed timing design required.**

Keywords— **Asynchronous VLSI; arbitration; metastability; synchronization failure; interrupts; quasi delay-insensitive**

I. Introduction

This paper introduces solutions to the problem of sampling a digital signal whose value may change at any time without waiting for an acknowledgment from the sampling circuit. In the asynchronous literature, a circuit that samples such a signal is called a *synchronizer*. This problem appears deceptively simple, yet no entirely satisfactory solution has been presented until now.

The synchronizers that we describe are more powerful devices than the synchronizers used in clocked systems or the well-known arbiters (mutual-exclusion elements) used in asynchronous systems. Clocked synchronizers (cascaded latches or flip-flops) are able to sample asynchronous inputs whether those inputs are stable (i.e., require an acknowledgment between each change in value) or not with a small but finite probability of synchronization failure, which might be catastrophic. Asynchronous arbiters that select between two stable inputs, on the other hand, may be designed with zero probability of catastrophic system failure. What we are discussing in this paper is a device that goes beyond the standard asynchronous arbiter to sample *arbitrary, unstable* inputs, while maintaining zero probability of system failure (as long as the components are reliable). The only previously known circuits that accomplish this task have several drawbacks: they are difficult to design and they consume static power.

In an event-driven asynchronous system, circuit modules are allowed to take as much time as they please to make their decisions. This property is what allows an asynchronous arbiter to avoid synchronization failure. However, if the input that is to be sampled can change from **true** to **false** and vice versa at any time, the arbiter can still glitch, and this has been an unsolved problem. For instance, Marshall *et al.* are faced with an asynchronous request signal that may be withdrawn at any time [5] and attempt to sample it with an arbiter. Their solution is to perform a hard system reset if the request is withdrawn.

Mika Nyström and Alain J. Martin are with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125, U.S.A. Rajit Manohar was with the Computer Science Department of the California Institute of Technology, Pasadena, CA 91125; he is now with the Computer Systems Laboratory in the School of Electrical Engineering at Cornell University, Ithaca NY 14853, U.S.A.

The structure of this paper is as follows. We first give a brief historical overview of the problem. Secondly, we specify the properties of a synchronizer, especially how they relate to the relative timing of the control signal and the input signal. Thirdly, we describe an older solution to the problem and expose its flaws. Fourthly, we describe a top-down solution to the problem that is efficient but depends on relative timing information related to the physics of the circuit elements. Fifthly, we present a conservative solution that is correct given a liberal delay model for the components. Sixthly, we present some variations on the conservative design. Lastly, we show some simulation results.

A. History

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 [1]. Synchronous designers must accept a certain risk of system failure, which can be traded against performance, as discussed in the literature [12].

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 are separated by at least a p-transistor threshold voltage—which means that

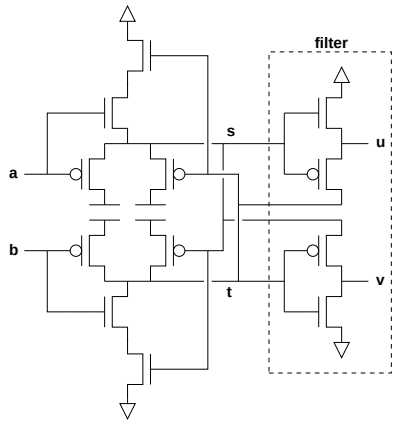

Fig. 1. "Mead & Conway" CMOS arbiter.

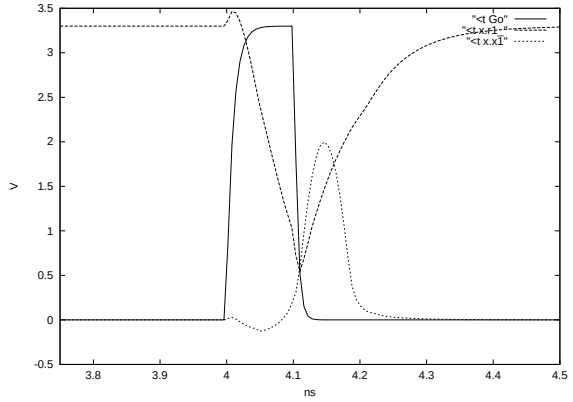

Fig. 2. Waveforms of a misbehaving arbiter.

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 waits 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 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), and this is the source of Marshall's conundrum. 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.

Expressed as a Production Rule Set (PRS)[9], the simple 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. This circuit exhibits metastability when both a and b are **true** simultaneously. A filtering prevents the outputs from changing until the arbiter has left the metastable state.

The arbiter is specified by the following *handshaking expansion* (HSE)[9]

$$\begin{aligned} *[&[a \longrightarrow u\uparrow; [\neg a]; u\downarrow \\ &|\, b \longrightarrow v\uparrow; [\neg b]; v\downarrow \\ &]] \end{aligned}$$

B. The QDI Model

Although a comprehensive survey of the different asynchronous design styles is outside the scope of this paper, we shall make reference to the assumptions satisfied by one of these styles, namely the *quasi delay-insensitive* (QDI) design style. (A more comprehensive survey has been carried out by Hauck [10].) A QDI system is one whose correct operation does not depend on the physical delay of operators (active elements) or wires, with the exception of certain wires called "isochronic forks" that are arranged so as to be fast compared to the circuitry with which they communicate [11]. One of the authors has shown that without the isochronic fork assumption, the class of circuits that is possible to prove correct is severely limited [11]; with the isochronic fork assumption, large asynchronous systems have been built, and their performance is excellent [6].

Violations of the isochronic fork assumption can occur in two ways. The first is, obviously, that an isochronic fork may extend over a large distance, and the signals at the "tines" of the fork are consequently delayed by different amounts. In large systems, we avoid making the isochronic fork assumption on long wires, although we sometimes find that the extra cost of verifying the assumption is outweighed by a more convenient circuit implementation (for example, in the case of busses). The second way that the isochronic fork assumption can be violated is more sinister. In any digital system, the logic threshold of every operator will not be the same as that of every other operator—this is particularly the case if early-threshold gates, due to dynamic circuit implementations, are mixed with combinational logic, as is often true in asynchronous systems. We must therefore ensure that logic transitions on isochronic forks are fast compared with the delay of the gates attached to the outputs of the fork.

C. The 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 **true** or **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.

The difficulty in the implementation of the synchronizer lies with the behavior when the input changes during the confusion interval. In fact, the specification allows the input to be an arbitrarily narrow pulse, among other disconcerting signals. All that is really required of the input is that it be within the range of safe device operation to prevent the circuit from being damaged.

D. Motivation

One may wonder why it would ever be necessary to have to sample a completely unsynchronized external signal. After all, if the signal changes, the synchronizer picks an arbitrary value, and an input value may be forever lost—how can this possibly be a useful behavior? The answer is that the ability to handle a withdrawn request is sometimes part of a specification. The MIPS ISA, for instance, allows an external interrupt request to be withdrawn at any time [2]. For this reason, a synchronizer circuit similar to the ones presented in this paper was used to implement the interrupt mechanism in the MiniMIPS processor designed at Caltech. Also, a mechanism similar to the sequenced withdrawal of requests mentioned in Section III-A was used to implement the arbitrated exception mechanism of the MiniMIPS processor. The MiniMIPS processor was successfully fabricated in 1998.

E. Our previous solution

One of the authors has previously presented a solution to the synchronizer problem [7]. This circuit has been used in successful chip projects, but alas, it is incorrect. The reason it is incorrect is subtle, and the analysis used to uncover the problem will turn out to be useful later. The PRS for the incorrect synchronizer is [9]:

$$
\begin{array}{rcl}
x & \mapsto & x\_\downarrow \\
\neg x & \mapsto & x\_\uparrow \\
x \wedge re \wedge t & \mapsto & s\downarrow \\
x\_ \wedge re \wedge s & \mapsto & t\downarrow \\
\neg re \vee \neg t & \mapsto & s\uparrow \\
\neg re \vee \neg s & \mapsto & t\uparrow
\end{array}
$$

This synchronizer is incorrect because it lacks a metastable state. In scenarios in which x changes close to the time when the sampling request on re arrives, we know that the synchronizer must enter a metastable state, since otherwise we would have built a bounded-delay synchronizer, and this is impossible [4]. If we consider the situation when $x =$ **false** and $re =$ **true**, we can simplify the production rules to:

$$
\begin{array}{rcl}
s & \mapsto & t\downarrow \\
\neg t & \mapsto & s\uparrow \\
\neg s & \mapsto & t\uparrow
\end{array}
$$

At first glance, this seems good. Clearly, the only stable state for this PRS is $s =$ **true**, $t =$ **false**, which is the desired outcome when x is **false**. However, if the device has only one stable state, then calculus tells us that it cannot have a metastable state, and thus it cannot possibly be a synchronizer.

We can go a step further and determine in what way the "synchronizer" fails. The production rules prevent it from producing an illegal result, but we have shown it to be incorrect. The only way to reconcile these facts is that the device enters a deadlocked state under some circumstances. To see how this may happen, assume that x is true when re is asserted—s will begin to fall. If x changes to **false** before s has fallen all the way, then t will begin to fall, but if s has fallen far enough before this happens, then the device may get stuck in a state in which both s and t are in the intermediate range between a legal logic zero and a legal logic one. (Please note that this deadlocked state is *not* the same as a metastable state: a metastable state is one that is dynamically unstable but that might hold for an unknown, although usually short, period of time; in contrast, this deadlocked state, once entered, cannot be exited.)

F. The solution of Rosenberger et al.

In their 1988 paper on Q-Modules, Rosenberger, Molnar, Chaney, and Fang presented a synchronizer circuit that satisfies the specification we have given here. Their design uses what is essentially a sense amplifier built out of two inverters as a central element. The inverter outputs are perturbed by the input signals, and the outputs are then filtered through the same kind of circuit used in the CMOS

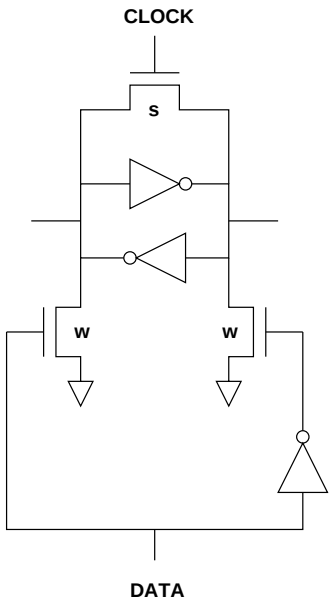

Fig. 3. Synchronizer of Rosenberger *et al.*

arbiter. The central part of Rosenberger *et al.*'s solution is shown in Figure 3 (the filtering circuit is not shown): the input is applied at $DATA$; the input is sampled on a falling edge of $CLOCK$. The problem with this circuit should be immediately evident: for it to operate correctly, certain transistors—nFETs marked w in the diagram—must be weak compared to the sense amp inverters' pFETs; another transistor—an nFET marked s in the diagram—must be strong. Setting up the required transistor-strength ratios is difficult; guaranteeing that they hold over a range of manufacturing and operating parameters is almost impossible [3]: Rosenberger *et al.* dedicate several pages to this matter. Finally, this circuit also draws static power when the $CLOCK$ is active. In contrast, the solutions studied in the present paper neither have ratioing assumptions nor draw static power.

II. 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:

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

whose environment is described by

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

From the metastability analysis of our previous, direct implementation of $SYNC$, we conclude that we cannot arbitrate between the two conditions $re \wedge x$ and $re \wedge \neg x$. Instead, 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 $a1$:

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

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

$$\begin{array}{l} SYNC2 \equiv *[[re \wedge \neg x \longrightarrow a0\uparrow; [\neg re]; a0\downarrow \\ \qquad\quad |\, re \wedge x \longrightarrow a1\uparrow; [\neg re]; a1\downarrow \\ \qquad]] \end{array}$$

$$\begin{array}{l} SEL \equiv *[[a0 \longrightarrow r0\uparrow; [\neg a0]; r0\downarrow \\ \qquad []\, a1 \longrightarrow r1\uparrow; [\neg a1]; r1\downarrow \\ \qquad]] \end{array}$$

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

(The bar "[]" means that both $a0$ and $a1$ cannot be high at the same time when SEL is executing the selection statement.) We would 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:

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

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

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 circuit considerations that enable this assumption are discussed in Section II-A.

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:

$$\begin{array}{l} SEL \equiv *[[a0 \longrightarrow r0\uparrow; [\neg a0 \wedge \neg a1]; r0\downarrow \\ \qquad\quad |\, a1 \longrightarrow r1\uparrow; [\neg a1 \wedge \neg a0]; r1\downarrow \\ \qquad]] \end{array}$$

(Note that we have re-introduced the "|" indicating that the selection between $a0$ and $a1$ is non-deterministic.) SEL

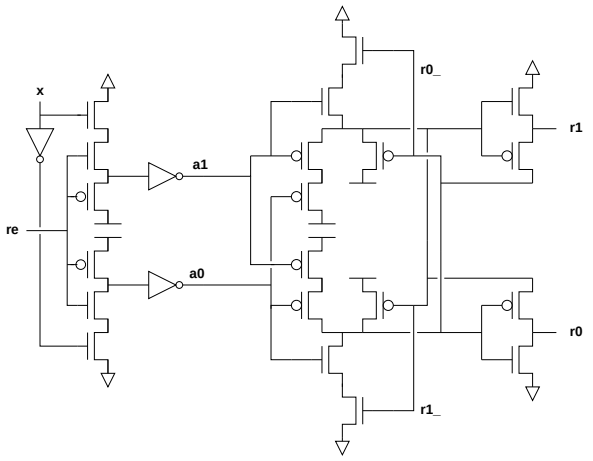

Fig. 4. Synchronizer

can be implemented directly as a bistable device, and the production rules are:

$$r1\_ \wedge a0 \mapsto r0\_\downarrow$$
$$\neg r1\_ \vee (\neg a0 \wedge \neg a1) \mapsto r0\_\uparrow$$

$$r0\_ \wedge a1 \mapsto r1\_\downarrow$$
$$\neg r0\_ \vee (\neg a1 \wedge \neg a0) \mapsto r1\_\uparrow$$

The circuit corresponding to the production rule set for the synchronizer is shown in Figure 4, where we have added the filter stage necessary to block the metastable state from reaching the digital outside world.

A. Analysis

Recall that the circuit discussed in Section I-E did not exhibit metastability, and that this led to incorrect operation. We examine the behavior of the circuit shown above under similar conditions.

When both $a0$ and $a1$ are **true**, the production rules reduce to:

$$r1\_ \mapsto r0\_\downarrow$$
$$\neg r1\_ \mapsto r0\_\uparrow$$

$$r0\_ \mapsto r1\_\downarrow$$
$$\neg r0\_ \mapsto r1\_\uparrow$$

This is a pair of cross-coupled inverters, a bistable device that exhibits metastability. Therefore, we do not have the same problem as the circuit in Section I-E.

Finally, we have to check whether the pullup/pulldown pairs for $r0\_$ and $r1\_$ have to be staticized. Let us do the analysis for $r0\_$. Starting in the initial state, $r0\_$ is pulled down by $r1\_ \wedge a0$; $r0\_$ can be left floating in the state $r1\_ \wedge \neg a0 \wedge a1$. But in this state $\neg re$ holds, and therefore the transition $a1\downarrow$ is enabled. Hence the floating state does not persist, and staticizers are not needed.

The input x does not have to be stable (i.e., it can oscillate arbitrarily, not just transition once) since the condition that $a0$ and $a1$ be pulled up *monotonically* still holds.

While the metastability analysis shows that process *SEL* cannot fall victim to the same problem as circuit in Section I-E, *SEL* has a still more subtle analog problem. If we are to satisfy the requirements that the circuit be correct under the stringent assumptions of the QDI design style, we find that there is a problem when resetting the device. In simple terms, if, e.g., the output $r0$ is asserted, we know that the input $a0$ must have been asserted at some time in the past. However, we do not know the voltage of $a1$ when $r0$ is observed to go high. We cannot tell if $a1$ is, perhaps, in the middle of the voltage range—it could be at a voltage low enough to allow $r0$ to reset, yet high enough to allow $r1$ to become active later—which might allow a glitch to appear on the $r1$ output. We note that this is only an issue if the resetting of $a1$ by the p-transistor gated by re is slow. If we feel confident that that will never be the case, we can ignore this problem, but if we insist that the circuit be QDI, the problem persists. (Recall that the QDI condition, defined in Section I-B, does not allow the correct operation of the circuit to depend on the delay of an operator.) For instance, adding Schmitt triggers on the inputs to *SEL* is not enough since the sufficient conditions (described in Section III) that allow the Schmitt triggers to have clean outputs do not obtain.

III. The Safe Synchronizer

We finish our quest for a reliable synchronizer by curing the timing assumption in the *SEL* process. The problem with the previous design is that $a0$ and $a1$ could have persistent intermediate values between a legal logic zero and a legal logic one. The cure is to ensure that both $a0$ and $a1$ are at a well-defined value, **true**, before resetting the circuit. The values $re \wedge x$ and $re \wedge \neg x$ are sampled as before and *SEL* is used to determine which of $r0$ or $r1$ should be asserted.

The safe synchronizer works as follows. Once the *SEL* process has decided which input is high, the *other* input is brought high. Once all inputs are in a known digital state, the circuit resets. This design isolates the state in which $a0$ and $a1$ can have intermediate values, allowing us to analyze the circuit in the digital domain. The digital analysis is sound as long as the inputs to the arbiter are monotonically increasing. Once the arbiter has decided that one of the inputs has become high, it will not "change its mind." If the input x changes while re is high, both $a0$ and $a1$ may rise, and the arbiter picks non-deterministically, as before. Applying our new approach to the program for the synchronizer, we introduce two new variables $x0$ and $x1$ and write the behavior of the circuit as

$$\begin{array}{l} *[[re \wedge \neg x \longrightarrow \\ \quad a0\uparrow; x0\uparrow; a1\uparrow; r0\uparrow; [\neg re]; a0\downarrow, a1\downarrow; x0\downarrow; r0\downarrow \\ \;|\, re \wedge x \longrightarrow \\ \quad a1\uparrow; x1\uparrow; a0\uparrow; r1\uparrow; [\neg re]; a0\downarrow, a1\downarrow; x1\downarrow; r1\downarrow \\ ]] \;. \end{array}$$

We make *SEL* drive $x0$ and $x1$ instead of the output directly and write

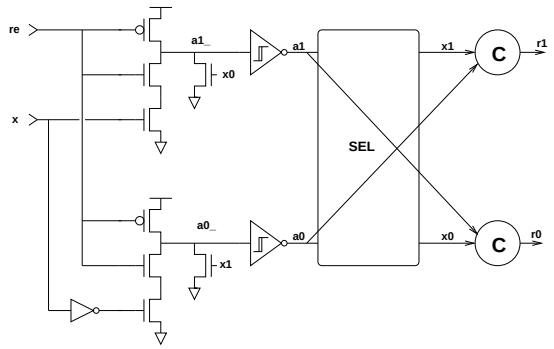

Fig. 5. Safe synchronizer.

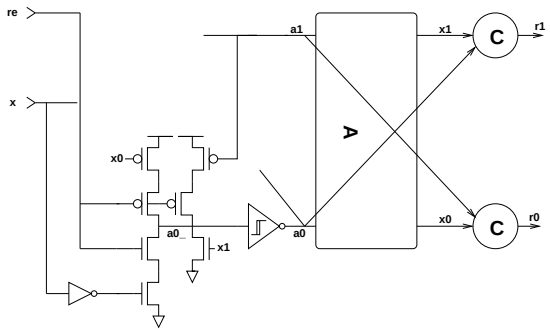

Fig. 6. Safe synchronizer built around standard arbiter (partial).

$$SEL \equiv *[[a0 \longrightarrow x0\uparrow; [\neg a0 \wedge \neg a1]; x0\downarrow \\ \;\;|\, a1 \longrightarrow x1\uparrow; [\neg a1 \wedge \neg a0]; x1\downarrow \\ ]] \;.$$

The rest of the synchronizer consists of the input integrators and an output stage, which we compile directly as follows.

The output stage is simply compiled as a pair of C-elements.

$$\begin{aligned} x0 \wedge a1 &\mapsto r0\uparrow \\ \neg x0 \wedge \neg a1 &\mapsto r0\downarrow \\ x1 \wedge a0 &\mapsto r1\uparrow \\ \neg x1 \wedge \neg a0 &\mapsto r1\downarrow \end{aligned}$$

The CMOS realization of these C-elements is with inverting C-elements followed by inverters.

The input integrators are, as before:

$$\begin{aligned} re \wedge \neg x \vee x1 &\mapsto a0\uparrow \\ \neg re &\mapsto a0\downarrow \\ re \wedge x \vee x0 &\mapsto a1\uparrow \\ \neg re &\mapsto a1\downarrow \end{aligned}$$

In the CMOS realization we have

$$\begin{aligned} re \wedge \neg x \vee x1 &\mapsto a0\_\downarrow \\ \neg re &\mapsto a0\_\uparrow \\ re \wedge x \vee x0 &\mapsto a1\_\downarrow \\ \neg re &\mapsto a1\_\uparrow \\ a0\_ &\mapsto a0\downarrow \\ \neg a0\_ &\mapsto a0\uparrow \\ a1\_ &\mapsto a1\downarrow \\ \neg a1\_ &\mapsto a1\uparrow \end{aligned}$$

But one issue remains. The production rules for $a0$ and $a1$ are unstable even though we have guaranteed that they will switch monotonically from **false** to **true** and back. By implementing the CMOS production rules for $a0$ and $a1$ as Schmitt triggers instead of inverters, we guarantee that $a0$ and $a1$ will now switch quickly and cleanly between the rails without glitches. A circuit diagram of the safe synchronizer is shown in Figure 5.

Readers experienced in asynchronous circuit design will point out that it is not enough that signals be monotonic; if a signal is changing monotonically but slowly, it is possible that a single value is interpreted as both a zero and a one logic value by different parts of a circuit. This is in essence due to the presence of isochronic forks where the transistors located at the tines of the fork have different voltage thresholds (see Section I-B). We use (inverting) Schmitt triggers to avoid this problem, using the property that an input that changes monotonically from one voltage rail to another generates an output that changes monotonically between the rails, but always fast enough to satisfy the isochronic fork assumption. The Schmitt trigger is a "slew rate amplifier." As a bonus, the Schmitt triggers add a satisfying noise margin to the design owing to their hysteresis. (Of course, this all comes at the cost of late, hysteretic thresholds and consequently longer latency.) The reason that we did not use Schmitt triggers in the synchronizer described in Section II is that $a0$ and $a1$ may have glitches in that circuit—an inertial delay would be required to remove those glitches.

A. Using a standard arbiter in a synchronizer

Sometimes, it is desirable to keep the number of "black boxes" (especially containing metastability) that need to be verified to a minimum. If we replace *SEL* in the safe synchronizer and ensure that $a0$ and $a1$ reset in the correct order, we can use a standard arbiter in place of *SEL* in the synchronizer.

Consider the program

$$*[[re \wedge \neg x \longrightarrow \\ \;\; a0\uparrow; x0\uparrow; a1\uparrow; r0\uparrow; [\neg re]; a1\downarrow; a0\downarrow; x0\downarrow; r0\downarrow \\ \;|\, re \wedge x \longrightarrow \\ \;\; a1\uparrow; x1\uparrow; a0\uparrow; r1\uparrow; [\neg re]; a0\downarrow; a1\downarrow; x1\downarrow; r1\downarrow \\ ]] \;;$$

the only difference between this program and that in the previous section is the sequencing of the downgoing actions on $a0$ and $a1$. By withdrawing the losing request before the winner, we can use a standard arbiter without introducing any glitches. To accomplish this, we have the PRS

$$\begin{aligned} re \wedge x\_ \vee x1 &\mapsto a0\_\downarrow \\ \neg re \wedge \neg a1 &\mapsto a0\_\uparrow \\ re \wedge x \vee x0 &\mapsto a1\_\downarrow \\ \neg re \wedge \neg a0 &\mapsto a1\_\uparrow \end{aligned}$$

with the rest of the circuit as before. A partial circuit diagram is shown in Figure 6. This design depends on the fact that the outputs of the arbiter will not switch when the second input is asserted.

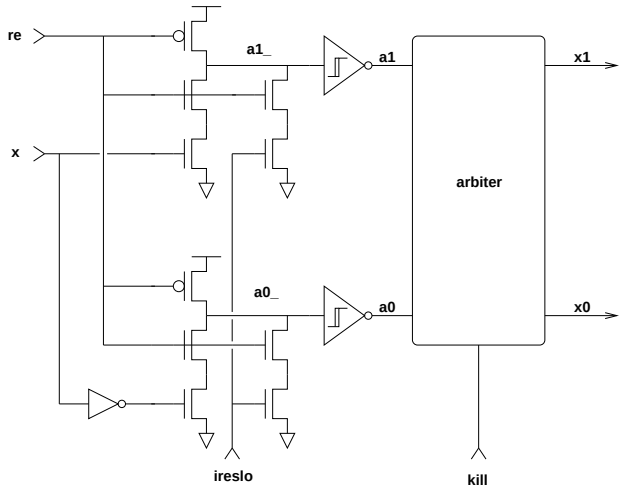

Fig. 7. Arbitration stage of synchronizer with *kill*.

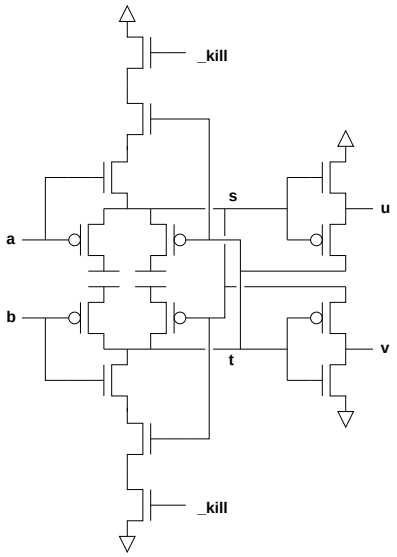

Fig. 8. Killable arbiter.

B. Version with "killable arbiter"

We have explored a number of other variations on the safe synchronizer. In our original design, we did not have the *SEL* process. In its place, we used a "killable" arbiter which allows resetting the inputs concurrently as long as a special *kill* signal is enabled. This design requires more complex control than the versions we have presented so far. The main advantage of the killable arbiter design is that both the winning and losing input are pulled up equally with the arbiter completely disabled. We no longer need to analyze the situation when two inputs to the arbiter are asserted specially.

The circuitry necessary to generate the *ireslo* and *kill* signals and to make the circuit obey the same I/O specification as the previous design is specified by the following HSE:

$$
\begin{array}{l}
*[\ [x0 \vee x1];\ \ ireslo\uparrow;\ \ [a0 \wedge a1];\ \ kill\uparrow; \\
\quad [x0 \longrightarrow r0\uparrow [] x1 \longrightarrow r1\uparrow]; \\
\quad [\neg a0 \wedge \neg a1];\ \ kill\downarrow;\ \ [\neg x1 \wedge \neg x0];\ \ ireslo\downarrow;\ \ r0\downarrow, r1\downarrow \\
\ ]
\end{array}
$$

We omit the straightforward production-rule compilation of this HSE, which results in four Muller C-elements and an or-gate.

The killable arbiter (Figure 8) is used in this design. When *_kill* is low, the arbiter cannot drive any of its outputs high, and the inputs to the arbiter may be reset in any order. The outputs will reset first when the kill signal is de-asserted.

IV. Performance measurements

None of the designs presented in this paper has yet been fabricated. We produced layout using Berkeley's Magic layout editor, using design rules for the MOSIS/HP 0.6-μm process. The layout was simulated using the Aspice analog simulator with BSIM2 parameters. Our parameters are known to over-state circuit performance by some 15–20 %, and the fact that our simulated layout does not include wiring loads would result in a further performance loss of about the same magnitude in a real implementation.

Implementation	Fast	Safe
Mean cycle (ns)	1.36	3.41
Standard deviation (ns)	0.031	0.306

Fig. 9. Aspice simulation results.

We connected each synchronizer to a simple process that repeatedly requests values (this is realized with a nor-gate and two inverters). The input signal x was taken from an unsynchronized eleven-stage ring oscillator running at approximately 300 MHz. We measured the interval between adjacent synchronizations (this includes any metastability due to inputs arriving within the confusion interval). The "fast" synchronizer was the design of Section II and the "safe" synchronizer was the version with the "killable arbiter" of Section III-B. The results are summarized in Figure 9.

The reader is cautioned that the performance figures in Figure 9 are in no sense absolute. The synchronizers presented in this paper have been designed to be understandable and not with performance in mind. Particularly the safe synchronizer could be improved by rearranging the signal senses. (The large standard deviation of the synchronization interval of the safe synchronizer is due to the fact that sometimes, because of the rapidly changing input, $a1$ and $a0$ are both asserted before *ireslo* goes high.) Any synchronizer design can also be improved simply by interleaving between two or more identical synchronizers. Still, the performance figures give an indication of the relative speeds of the two designs.

In Figures 10–14 we show an example of two synchronizations performed by the fast synchronizer. Figure 10 shows the inputs x, $x\_$, and the request input *re*. Figure 11 shows the outputs $a0\_$ and $a1\_$ of the "integrators," Figure 12 shows the inputs $a0$ and $a1$ to *SEL*, the arbitra-

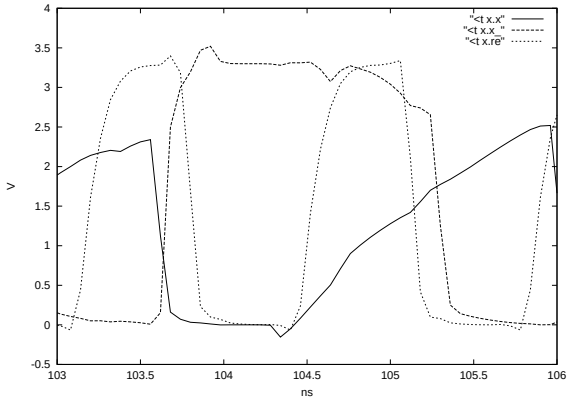

Fig. 10. Synchronizer inputs.

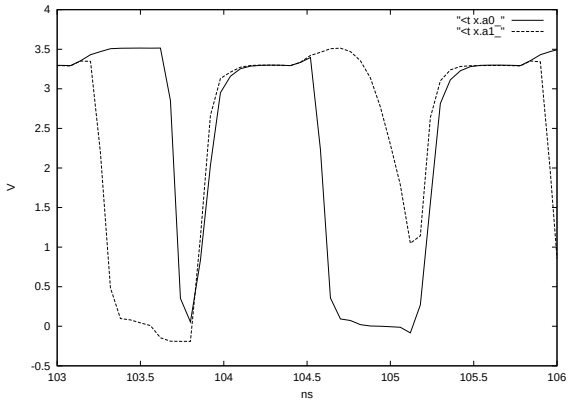

Fig. 11. Integrator outputs.

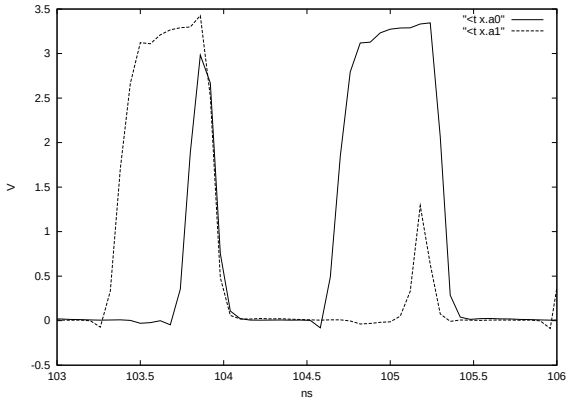

Fig. 12. *SEL* inputs.

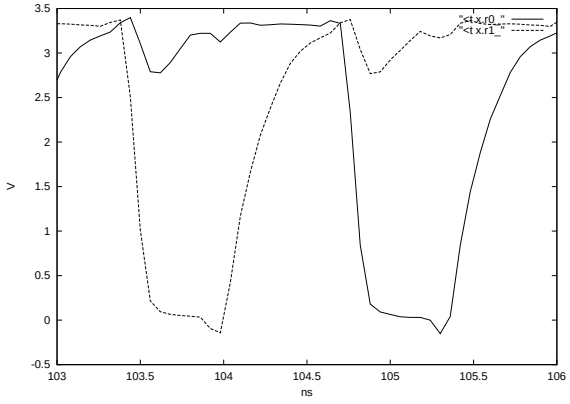

Fig. 13. Internal nodes.

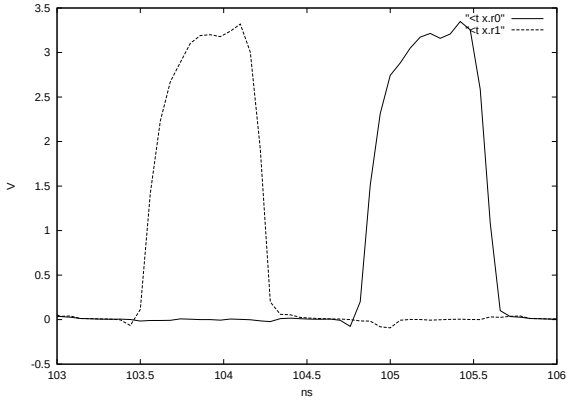

Fig. 14. Synchronizer outputs.

tion part of the synchronizer, Figure 13 shows the internal nodes $r0\_$ and $r1\_$ of the arbitration stage, and finally Figure 14 shows the clean outputs $r0$ and $r1$ of the device.

V. Conclusion

We have described a synchronizer that is guaranteed to output a legal digital dual-rail value that represents the value of an unsynchronized input in some bounded time neighborhood of the synchronization request. Our circuit depends on the usual assumptions of quasi delay-insensitive design (namely, the concept of isochronic fork) as well as on certain monotonicity properties of the signals, but no further timing assumptions or restrictions on the environment are necessary.

The two designs we have presented have different strengths. The first synchronizer of Section II is a simple, fast circuit, but unfortunately it depends on a timing assumption, which can be a drawback in some situations. The safe designs, on the other hand, are quasi delay-insensitive, but they add extra complexity in the form of Schmitt triggers and control circuitry, and they are slower.

The synchronizer design problem reveals the importance of considering subtle timing issues when dealing with

metastable circuits. While it is probably true that the "unsafe" synchronizer we have presented is perfectly safe (i.e., will not malfunction) under any reasonable device parameters and input scenarios, a naïve analysis of the circuit in terms of digital production rules might lead to the incorrect conclusion that it is quasi delay-insensitive.

VI. Acknowledgments

The research described in this report was sponsored by the Defense Advanced Research Projects Agency and monitored by the Office of Army Research. Mika Nyström was supported in part by an Okawa Foundation fellowship and an IBM Research Fellowship. Rajit Manohar was supported in part by a National Science Foundation CAREER award. The authors thank Andrew Lines for convincing us that it is perhaps not necessary to bring all the signals to a known value between synchronizations and Paul Pénzes for suggesting that something was amiss with the previously known solution.

References

[1] 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.

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

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

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

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

[6] 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," in *Proceedings of the 17th Conference on Advanced Research in VLSI*. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[7] 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. Addison-Wesley, Reading, Mass., 1989.

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

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

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

[11] 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.

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

[13] 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.

[14] Charles L. Seitz. "System timing," chapter 7 in [12].