A Delay-insensitive Fair Arbiter

Alain J. Martin

Computer Science Department
California Institute of Technology

5193:TR:85

A Delay-Insensitive Fair Arbiter

Alain J. Martin

Computer Science Department
California Institute of Technology

5193:TR:85

The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

A Delay-Insensitive Fair Arbiter

Alain J. Martin
Computer Science
California Institute of Technology
Pasadena CA 91125
June 1985, May 1986

1. Introduction

In any device communicating with its environment by two—or more—independent channels, the problem arises of selecting one request for communication among several ones. This problem is called the "arbitration problem", and the part of the device that makes the selection is called an "arbiter". The main building-block of an arbiter is a so-called "basic arbiter" or "mutual exclusion" element, which is used for making a non-deterministic choice between two elementary input signals.

It is by now well-known that in any physical realization of a basic arbiter, the time the device takes to reach a stable state when both input signals are **true** i.e., when both input signals may be selected—is potentially unbounded because of the metastability phenomenon [1], [5]. However, this timing property of the basic arbiter does not cause any problem in a delay-insensitive, or self-timed (we use the two terms as synonyms) design discipline in which no timing assumptions are made on the delays in wires and operators, except that the delays are finite.

But the realization of a delay-insensitive arbiter raises another issue: that of fairness. An arbiter is *strongly fair* when a pending communication request is guaranteed to be granted after at most N other requests are granted, where N is a given positive integer constant. An arbiter is *weakly fair* when a request is granted after a finite number of other requests. Whether it is possible to construct a delay-insensitive fair arbiter has been, so far, an open question. It has been conjectured, [6], that delay-insensitive, fair arbiters do not exist. In this paper we prove the existence of delay-insensitive fair arbiters by constructing one.

We apply the synthesis method described in [3] and [4]. We first describe the arbiter in terms of a communicating process. It is easy to prove that this arbiter is fair. We then "compile" this program into a delay-insensitive circuit by applying a series of systematic, semantics-preserving transformations. Hence the circuit obtained is correct by construction, i.e. has the same fairness properties as the original program.

The choice of the operators used in the circuit will be discussed after constructing the arbiter. We use only standard operators: wire, fork, and-gate, or-gate, C-element, basic arbiter, and synchronizer. (An operator may have any number of inputs or outputs negated.) Furthermore we do not assume the basic arbiter or any other operator to be fair. All operators except the forks are delay-insensitive i.e. no assumption is made on the propagation delays in these operators.

The fork is the only operator with more than one output—in this paper we use only forks with two outputs—, which poses the additional problem of guaranteeing that a change of input value is followed by a corresponding change of value on *both* outputs of a fork. For reasons of simplicity, we solve this problem by assuming that the propagation delays in forks is short compared to the delays in all operators the fork can be connected to (all other operators except wires and forks). Forks with this property are said to be "isochronic".

2. A fair arbiter program

Processes communicate with each other by communication commands. Communication command X in process p is paired with communication command Y in process r by declaring the pair (X,Y) to be a channel between p and r. X and Y form a pair of synchronization primitives: the completion of the n*th* X-action in p coincides with the completion of the n*th* Y-action in r (synchronization requirement). From initiation until completion, an action is "pending" or "suspended". An action is suspended if and only if the completion of that action would violate the synchronization requirement (progress requirement).

We also provide a general Boolean command on channels, called the *probe*. In the original definition proposed in [2], given the channel (X,Y), the "probe on X", denoted by $\overline{X}$, has the same value as the predicate "a Y action is pending", denoted by qY. In the context of a delay-insensitive implementation, we use a weaker definition, namely:

$$\overline{X} \Rightarrow qY,$$

$$qY \Rightarrow \diamond\overline{X},$$

where $\diamond P$ means P *holds eventually*, i.e. P holds after a finite, but possibly unbounded, number of actions. (Either definition can be used, but the discussion about the fairness of the arbiter is clearer with the weaker definition.)

Given the above communication primitives, an arbiter can be described in its simplest form as a process R communicating with the environment by two independent channels (A,A') and (B,B'). The arbiter is a non-terminating process, each elementary step of which is either A or B, and which is suspended if and only if neither A nor B can be completed, i.e. neither A' nor B' is pending. Without taking fairness into account, a solution for R is

$$\begin{aligned} *[&[\overline{A} \rightarrow A \\ &|\overline{B} \rightarrow B \\ &]]. \end{aligned} \tag{1}$$

The construct $*[S]$ means "repeat S forever". The execution of the *selection* command $[G_1 \rightarrow S_1 | G_2 \rightarrow S_2]$, where G_1 and G_2 are Boolean expressions called the *guards*, and S_1 and S_2 are program parts, amounts to the execution of an arbitrary S_i for which G_i holds. (The expression $G_i \rightarrow S_i$ is called a *guarded command*). If none of the guards is **true** , the execution of the selection command is suspended until some guard is **true** . Since when several guards are **true** , the choice of the guarded command to be executed is arbitrary, the selection command is not fair. The choice of an unfair selection command is consistent with the implementation: in order to select one out of two true guards, we need to use a mutual exclusion element, and such an element is not fair.

Given the above semantics for the selection command, solution (1) is obviously not fair. There are several ways to transform the above solution into a fair one. They all require testing whether a certain communication action is pending, which is easy to do with the probe primitive. We choose to implement the following solution, suggested by Kevin Van Horn:

$$*[[\overline{A} \rightarrow A \mid \neg\overline{A} \rightarrow \mathbf{skip}]; [\overline{B} \rightarrow B \mid \neg\overline{B} \rightarrow \mathbf{skip}]] \qquad (2)$$

According to (2), when $\overline{A}$ holds, A will be completed after at most one B action, whatever the current state of R is. Hence, R is strongly fair towards requests A and B. Assume that A' is pending at a certain point of the computation. By definition of the probe, $\overline{A}$ is **true** eventually, i.e., a finite but unbounded number of B actions can be completed between the moment qA' holds and the moment $\overline{A}$ holds. Hence, the arbiter is only *weakly* fair towards requests A' and B'. Therefore, with this definition of the probe, we can say that the arbiter is strongly fair towards requests that have reached the arbiter and weakly fair towards all pending requests. With the stronger definition, a request is pending only when it has reached the arbiter, and therefore the arbiter is strongly fair towards all pending requests.

3. The "compilation" method

Next, we shall transform program (2) into a delay-insensitive circuit, according to the method described in [3]. The main steps of the method will be explained briefly as we proceed with the transformation of (2).

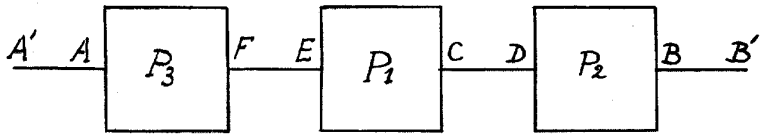
Process decomposition

Decomposition rule: A process P containing an arbitrary program part S is semantically equivalent to two processes $P1$ and $P2$, where $P1$ is derived from P by replacing S by a communication C on the newly introduced channel (C, D) between $P1$ and $P2$, and $P2 \equiv *[[\overline{D} \rightarrow S; D]]$.

Observe that the above decomposition does not introduce concurrency. Although $P1$ and $P2$ are potentially concurrent processes, they are never active concurrently: $P2$ is activated from $P1$ as a procedure or a coroutine would be. The only purpose of this transformation is to simplify the structure of each command.

Applying this decomposition rule, we decompose (2) into three processes ($P1 \parallel P2 \parallel P3$). Channels (C, D) between $P1$ and $P2$, and (E, F) between $P1$ and $P3$ are introduced. (See figure 1.)

$$
\begin{aligned}
P1 &\equiv *[\,E;C\,].\\
P3 &\equiv *[\,[\overline{F} \wedge \overline{A} \rightarrow A;F\\
&\quad\ \ |\overline{F} \wedge \neg\overline{A} \rightarrow F\\
&\quad\ ]\,].\\
P2 &\equiv *[\,[\overline{D} \wedge \overline{B} \rightarrow B;D\\
&\quad\ \ |\overline{D} \wedge \neg\overline{B} \rightarrow D\\
&\quad\ ]\,].
\end{aligned}
$$



–Figure 1–

Handshaking expansion

The next step, called "handshaking expansion", replaces each channel by a pair of wire-operators and each communication action by its four-phase handshaking implementation. Channel (X,Y) is implemented by the two wires $(xo\ \underline{w}\ yi)$ and $(yo\ \underline{w}\ xi)$. The communication actions on (X,Y) can be implemented either with X "active" and Y "passive" as follows:

$$X \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \tag{3}$$

$$Y \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow \tag{4}$$

or vice-versa with X passive and Y active. Initially, all variables are **false**. A probed communication action $\overline{X} \rightarrow \ldots X$ must be implemented:

$$xi \rightarrow \ldots xo\uparrow;\ [\neg xi];\ xo\downarrow. \tag{5}$$

Hence a probed action is implemented as passive and the matching Y-action must then be implemented as active.

Production-rule expansion

The next step consists in replacing the set of guarded commands obtained after handshaking expansion by an equivalent set of guarded commands in which all explicit sequencing (every semicolon) has been removed. Such a guarded command is called a "production rule". The problem is to define the guard of each rule such that the firing sequence of the rules is equivalent to the execution of the original program. In this transformation step, an important property of the four-phase handshaking is used, namely that the first and third semicolons of (3) and the second semicolon of (4) need not be implemented: the sequencing is enforced by the protocol.

Operator reduction

The last step consists in identifying sets of production rules in the program with sets of production rules describing the semantics of operators. The operators and their production rule specifications are given in the appendix. The program can then be identified with a network of operators. $P1$, $P2$, $P3$ will now be compiled in sequence.

4. Compilation of P1

Since commands D and F are probed, they have to be implemented as passive, and thus commands C and E have to be implemented as "active". Hence $P1$ is the standard "AA-adaptor" circuit [4]. The handshaking expansion of $P1$ gives:

$$P1 \equiv *[eo\uparrow;\ [ei];\ eo\downarrow;\ [\neg ei];\ co\uparrow;\ [ci];\ co\downarrow;\ [\neg ci]]$$

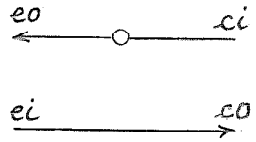
The first—and easiest—way to compile $P1$ is to reshuffle some actions. For instance, we can postpone the sequence $eo\downarrow;\ [\neg ei]$ until after $[ci]$. Since this reshuffling introduces some synchronization between E and C, and thus between A and B, it can be performed only if the actions A' and B' are independent. (Otherwise the synchronization introduced could be incompatible with the synchronization between A' and B' and could introduce deadlock.) We get:

$$P1 \equiv *[eo\uparrow;\ [ei];\ co\uparrow;\ [ci];\ eo\downarrow;\ [\neg ei];\ co\downarrow;\ [\neg ci]]$$

The production rule expansion is straightforward:

$$\begin{aligned} \neg ci &\mapsto eo\uparrow \\ ei &\mapsto co\uparrow \\ ci &\mapsto eo\downarrow \\ \neg ei &\mapsto co\downarrow \end{aligned}$$

And the circuit is simply one inverter and one wire as shown in figure 2.



–Figure 2–

The compilation of $P1$ without reshuffling requires introducing a state variable u:

$$P1 \equiv *[eo\uparrow;\ [ei];\ u\uparrow;\ [u];\ eo\downarrow;\ [\neg ei];\ co\uparrow;\ [ci];\ u\downarrow;\ [\neg u];\ co\downarrow;\ [\neg ci]]$$

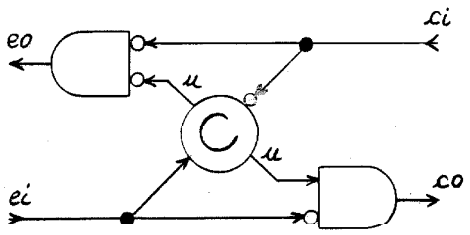
The production rule expansion gives:

$$\begin{aligned} \neg ci \wedge \neg u &\mapsto eo\uparrow \\ \neg ci \wedge ei &\mapsto u\uparrow \\ ci \vee u &\mapsto eo\downarrow \\ u \wedge \neg ei &\mapsto co\uparrow \\ ci \wedge \neg ei &\mapsto u\downarrow \\ \neg u \vee ei &\mapsto co\downarrow \end{aligned}$$

The operator reduction gives:

$$\begin{aligned} (\neg ci,\ \neg u) &\triangle eo \\ (\neg ci,\ ei) &\ \underline{C}\ u \\ (u, \neg ei) &\triangle co \end{aligned}$$

The circuit is shown in figure 3:



–Figure 3–

5. Compilation of P2

The handshaking expansion gives:

$$\begin{aligned} P2 \equiv *[[\ & di \wedge bi \rightarrow bo\uparrow; [\neg bi]; bo\downarrow; do\uparrow; [\neg di]; do\downarrow && (5.1)\\ & \mid di \wedge \neg bi \rightarrow do\uparrow; [\neg di]; do\downarrow && (5.2)\\ &]]. \end{aligned}$$

Because bi can change from **false** to **true** asynchronously, the second guard of $P2$ is not "stable", i.e. its value can change from **true** to **false** at any time. In order to make both guards of $P2$ stable, we have to replace bi and $\neg bi$ in the guards by the stable copies u and v, i.e. the values of u and v are the same as the values of bi and $\neg bi$ respectively, before the transition $di\uparrow$. When the transitions $bi\uparrow$ and $di\uparrow$ take place concurrently, u and v may take any values, provided $u \equiv \neg v$ holds. To this effect, we introduce the synchronizer $(bi, di)\ \underline{S}\ (u, v)$. $P2$ becomes:

$$
\begin{aligned}
P2 \equiv *[[\ & bi \wedge di \wedge \neg v \rightarrow u\uparrow && (5.3)\\
|\ & \neg bi \wedge di \wedge \neg u \rightarrow v\uparrow && (5.4)\\
|\ & \neg di \wedge u \rightarrow u\downarrow && (5.5)\\
|\ & \neg di \wedge v \rightarrow v\downarrow && (5.6)\\
|\ & u \rightarrow bo\uparrow; [\neg bi]; bo\downarrow; do\uparrow; [\neg di]; do\downarrow && (5.7)\\
|\ & v \rightarrow do\uparrow; [\neg di]; do\downarrow && (5.8)\\
]]\ .
\end{aligned}
$$

Commands (5.3) through (5.6) are the specification of the synchronizer; (5.7) and (5.8) are derived from (5.1) and (5.2) respectively, by replacing the guard of (5.1) by u and the guard of (5.2) by v. The rest of the compilation of $P2$ consists in compiling (5.7) and (5.8). The compilation of (5.7) is facilitated if transition $bo\downarrow$ is postponed until after $[\neg di]$. This transformation does not introduce deadlock since the completion of D does not depend on the completion of B. We also introduce the sequences $u\downarrow;[\neg u]$ in (5.7) and $v\downarrow;[\neg v]$ in (5.8) in accordance with (5.5) and (5.6) respectively. We obtain:

$$
\begin{aligned}
& u \rightarrow bo\uparrow; [\neg bi]; do\uparrow; [\neg di]; u\downarrow; [\neg u]; bo\downarrow; do\downarrow && (5.7)\\
& v \rightarrow do\uparrow; [\neg di]; v\downarrow; [\neg v]; do\downarrow\ . && (5.8)
\end{aligned}
$$

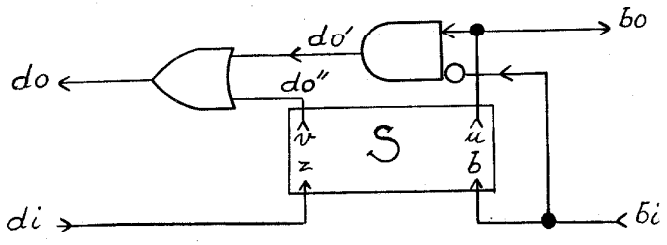
Which gives the production rules:

$$
\begin{aligned}
u &\mapsto bo\uparrow\\
u \wedge \neg bi &\mapsto do\uparrow\\
\neg di \wedge u &\mapsto u\downarrow\\
bi \vee \neg u &\mapsto do\downarrow\\
\neg u &\mapsto bo\downarrow\\
v &\mapsto do\uparrow\\
\neg di \wedge v &\mapsto v\downarrow\\
\neg v &\mapsto do\downarrow\ .
\end{aligned}
$$

The operator reduction gives

$$
\begin{aligned}
u \ \underline{w}\ & bo\\
(u, \neg bi) \ \underline{\triangle}\ & do'\\
v \ \underline{w}\ & do''\\
(do', do'') \ \underline{\vee}\ & do.
\end{aligned}
$$

The graph of the circuit is shown in figure 4:

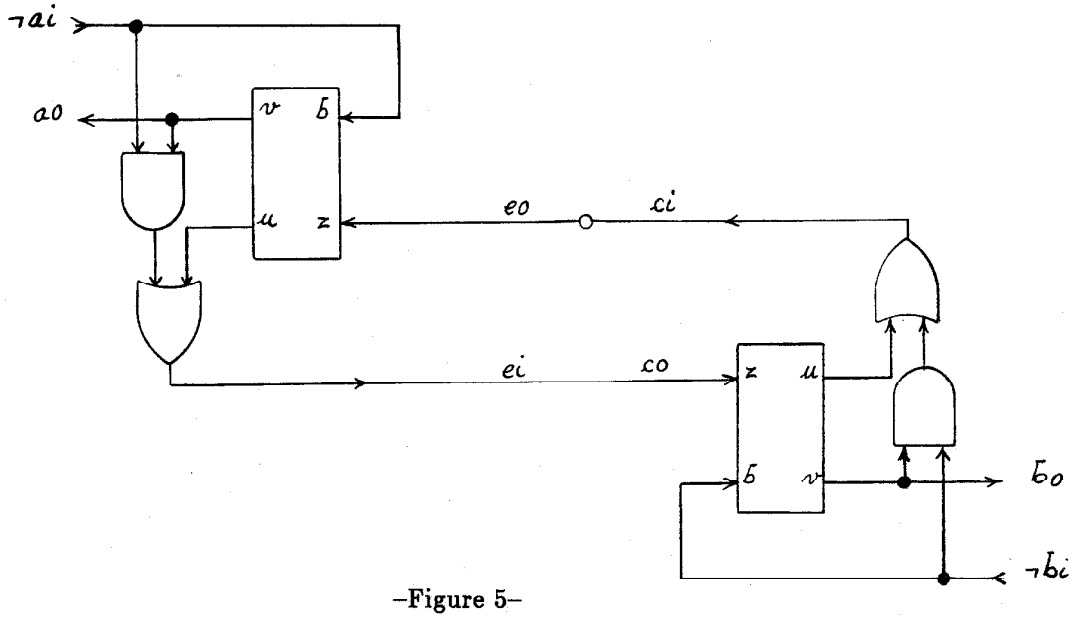


–Figure 4

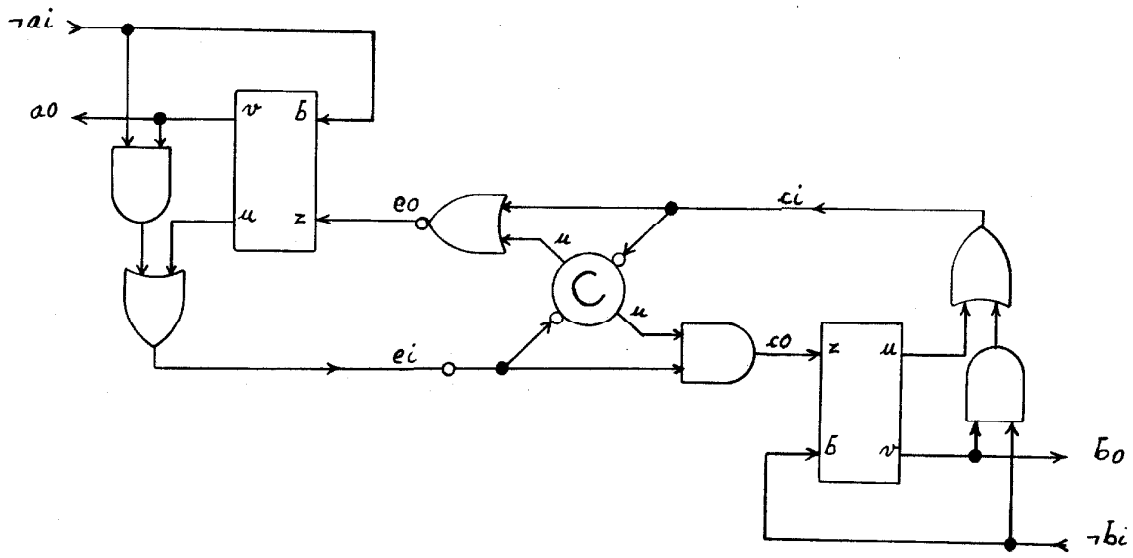
The implementation of P3 is identical.

6. The circuit

The final circuit is obtained by composing the two identical circuits implementing $P2$ and $P3$ by the circuit of $P1$. The simpler version of $P1$ gives the circuit of figure 5. The "quick-return linkage" implementation of $P1$ gives the circuit of figure 6. During the construction of those circuits, we have applied a minor optimization procedure: we have eliminated the negated inputs that are also the output of a fork. We leave it as an exercise to the reader to convince himself that the transformations do not change the semantics of the circuits.



–Figure 5–



–Figure 6–

7. Concluding remarks

We have constructed a circuit for fair arbitration by "compiling" a program into a network of logical operators. For each step of the transformation, it can be verified that the "object" program is equivalent to the "source" program. Hence the final circuit is semantically equivalent to the original program. The circuit is delay-insensitive, i.e. its correct interaction with the environment is independent of delays in wires and operators.

The circuit is particularly simple because we have deliberately destroyed the symmetry between A and B and because we allow busy waiting. Busy waiting can be eliminated by implementing the program:

$$*[[\overline{A} \vee \overline{B} \rightarrow [\overline{A} \rightarrow A \mid \neg\overline{A} \rightarrow \textbf{skip}];$$
$$[\overline{B} \rightarrow B \mid \neg\overline{B} \rightarrow \textbf{skip}]$$
$$]].$$

The implementation of this program requires a mutual exclusion element in order to evaluate $\overline{A} \vee \overline{B}$ without introducing a hazard.

It is interesting to notice that the solution we have constructed can be immediately generalized for an arbitrary number of requests.

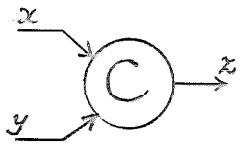
Acknowledgement This work benefited from many stimulating discussions with David Black, Martin Rem, Charles Seitz, Jan van de Snepscheut, and Kevin Van Horn. Acknowledgement is also due to W. Dally, Pieter Hazewindus, Blake Lewis, and Peggy Li for their comments.

Appendix

The operators used in the construction of the arbiter are the following.

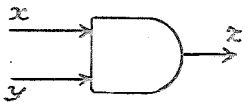
The C-element:

$$(x,y)\ \underline{C}\ z \equiv \quad x \wedge y \mapsto z\uparrow$$
$$\neg x \wedge \neg y \mapsto z\downarrow .$$



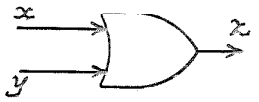
The "and":

$$(x,y)\ \underline{\wedge}\ z \equiv \quad x \wedge y \mapsto z\uparrow$$
$$\neg x \vee \neg y \mapsto z\downarrow .$$



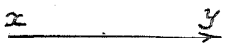
The "or":

$$(x,y)\ \underline{\vee}\ z \equiv \quad x \vee y \mapsto z\uparrow$$
$$\neg x \wedge \neg y \mapsto z\downarrow .$$



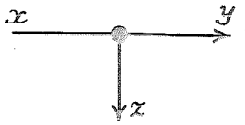
The wire:

$$x\ \underline{w}\ y \equiv \quad x \mapsto y\uparrow$$
$$\neg x \mapsto y\downarrow .$$



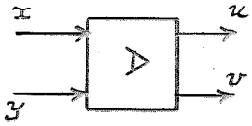
The fork:

$$x\ \underline{f}\ (y,z) \equiv \quad x \mapsto y\uparrow, z\uparrow$$
$$\neg x \mapsto y\downarrow, z\downarrow .$$



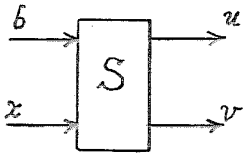
The basic arbiter (mutual exclusion element):

$$(x,y)\ \underline{A}\ (u,v) \equiv \quad x \wedge \neg v \mapsto u\uparrow$$
$$y \wedge \neg u \mapsto v\uparrow$$
$$\neg x \wedge u \mapsto u\downarrow$$
$$\neg y \wedge v \mapsto v\downarrow .$$



The synchronizer:

$$(b,z)\ \underline{S}\ (u,v) \equiv \quad b \wedge z \wedge \neg v \mapsto u\uparrow$$
$$\neg b \wedge z \wedge \neg u \mapsto v\uparrow$$
$$\neg z \wedge u \mapsto u\downarrow$$
$$\neg z \wedge v \mapsto v\downarrow .$$



References

1) Chaney, T.J. and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits", *IEEE Trans on Computers*, vol C-22, no.4, April 1973, pp 421-422

2) Martin, A.J., "The Probe: An Addition to Communication Primitives", *Information Processing Letters* 20, April 1985, pp 125-130

3) Martin, A.J., "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion", *Proc. 1985 Chapel Hill Conf. VLSI*, ed Henry Fuchs, pp. 247-260, March 1985

4) Martin, A.J., "Compiling Communicating Processes into Delay-insensitive VLSI circuits", to appear in *Journal of Distributed Computing*, vol.1, no.3, 1986.

5) Seitz, C.L., "System Timing", Chapter Seven in *Introduction to VLSI Systems* by Carver A Mead and Lynn A Conway, Addison-Wesley, 1980.

6) Udding, J.T., "On the Non-existence of Delay-insensitive Fair Arbiters", January 1985, private communication, JTU14.