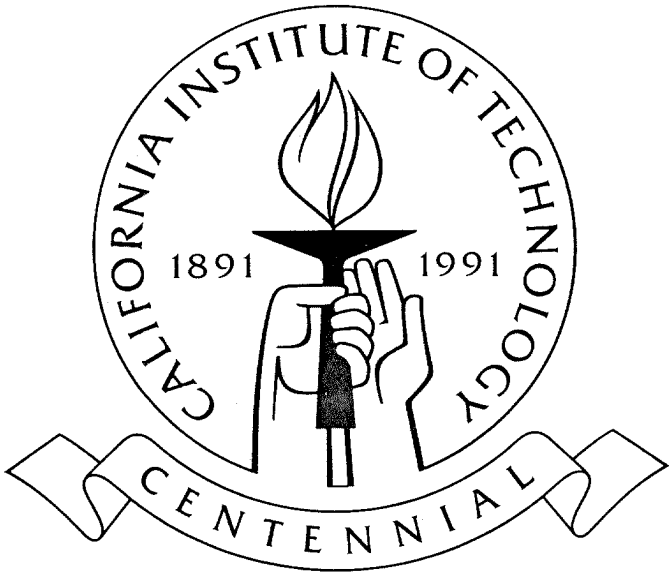


# Asynchronous Circuits for Token-Ring Mutual Exclusion

Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-90-09

# Asynchronous Circuits for Token-Ring Mutual Exclusion

**Alain J. Martin**

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, ARPA Order Number 6202; and monitored by the Office of Naval Research, under contract number N00014-87-K-0745.

Department of Computer Science
California Institute of Technology
Pasadena CA 91125

Caltech-CS-TR-90-09

# Asynchronous Circuits for Token-Ring Mutual Exclusion

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

18 June 1990

## 1 Introduction

In [1], we have described three algorithms for distributed mutual exclusion on a ring. All algorithms use a token to select a candidate. We have already implemented the most efficient of these algorithms as an asynchronous VLSI circuit. We are now going to implement the simplest one.

An arbitrary number ($> 1$) of cyclic automata, called "masters," make independent requests for exclusive access to a shared resource. The circuit should handle the requests from the masters in such a way that

1. any request is eventually granted, and
2. there is at most one master using the shared resource at any time.

The masters are independent of each other: They do not communicate with each other, and the activity of a master not using the resource should not influence the activity of other masters.

A master, $M$, communicates with its private server, $m$. When $M$ wants to use the shared resource ($M$ is said to be a *candidate*), it issues a request to $m$. When the request is accepted, $M$ uses that resource (for a finite period of time), and then informs $m$ that the resource is free again.

The servers are connected in a ring. At any time, exactly one (arbitrary) server holds a "privilege," or "token." The token circulates continuously around the ring of servers, and only the server that holds the token may grant the resource to its master, which guarantees mutual exclusion on the access to the resource.

The simplicity of the solution is due to the fact that we can encode the passing of the token between servers without introducing an explicit message or boolean variable. By definition, a server has the token if and only if it has completed a communication on its left channel $L$ and has not yet completed the following communication on its right channel $R$.

$$\begin{aligned} \mathit{master} &\equiv *[\ldots D; CS; D] \\ \mathit{server} &\equiv *[L; [\overline{U} \rightarrow U; U [] \neg \overline{U} \rightarrow \mathit{skip}]; R]. \end{aligned}$$

In order to start the ring with a token in one server, one server must be initialized in the state preceding $R$. In other words, it has to implement the sequence:

$$*[R; L; [\overline{U} \rightarrow U; U [] \neg \overline{U} \rightarrow \mathit{skip}]] \ .$$

# 2 Implementation of a Server Process

We first decompose a server into two processes by the usual decomposition technique. We get:

$$\begin{aligned} m1 &\equiv *[\, L; S; R] \\ m2 &\equiv *[\, [\overline{U} \wedge \overline{S} \rightarrow U; U; S \\ &\qquad [\, \overline{S} \wedge \neg \overline{U} \rightarrow S \\ &\quad ]] \end{aligned}$$

# 3 Compilation of m2

We start with the compilation of $m2$ since it will remained unchanged through all different compilations of the program. We implement the two consecutive $U$ communications as passive two-phase handshaking expansions, which is equivalent to replacing the two $U$ communications with one passive four-phase handshaking expansion.

Since $U$ can change from false to true at any time, the two guards of $m2$ can both be evaluated to true. We therefore need to introduce an arbiter or a synchronizer. Since we know that the basic arbiter and the basic synchronizer both require a four-phase protocol, we implement $S$ with a (passive) four-phase handshaking expansion. We get:

$$\begin{aligned} m2 \equiv *[\, [\, & si \wedge ui \rightarrow uo\uparrow; [\neg ui]; uo\downarrow; so\uparrow; [\neg si]; so\downarrow \\ & [\, si \wedge \neg ui \rightarrow so\uparrow; [\neg si]; so\downarrow \\ & ]] \ . \end{aligned}$$

The structure of the guards suggests that we introduce a synchronizer. It

is the standard process:

$$sync \equiv *[\,[si \wedge ui \rightarrow u\uparrow; [\neg si]; u\downarrow \\ \;\;|si \wedge \neg ui \rightarrow v\uparrow; [\neg si]; v\downarrow \\ \,]]\,,$$

in which $ui$ and $si$ are the variables of $m1$, and $u$ and $v$ are new auxiliary variables.

We now have to derive a process $m3$ such that $(m3||sync) = m2$. Since exactly the same decomposition has already be done in [5], we shall not repeat it. We get:

$$m3 \equiv *[\,[u \rightarrow uo\uparrow; [\neg ui]; uo\downarrow; so\uparrow; [\neg u]; so\downarrow \\ \;\;[]v \rightarrow so\uparrow; [\neg v]; so\downarrow \\ \,]]\,.$$

The compilation of the first guarded command is facilitated if the transition $uo\downarrow$ is postponed until after the wait $[\neg u]$. This transformation does not introduce deadlock since the completion of $U$ does not depend on the completion of $S$. It is important to observe that the whole use of the critical section takes place between $uo\uparrow$ and $[\neg ui]$ in $m3$. The rest of the compilation is also described in [5]. It gives the set of operators:

$$u \;\underline{w}\; uo$$
$$(u, \neg ui)\,\underline{\wedge} v'$$
$$(v, v')\,\underline{\vee} so$$

where $v'$ is an auxiliary variable. The circuit for $m2$ is shown in Figure 1.

# 4 Four-phase Implementation of m1

Process $m1$ is just the repetition of three communication actions in sequence. We choose to have $L$ passive and $R$ active, and $S$ has to be active because it is probed in $m2$. For reasons of efficiency, we slightly modify $m1$ as:

$$*[\overline{L} \rightarrow L \bullet S; R]\,.$$

If we ignore $S$, the process is just a standard "passive/lazy-active" buffer, which we have compiled in [6]. The handshaking expansion, including the handshaking sequence of $S$ and the state variable $x$, gives:

$$*[[li \wedge \neg si]; lo\uparrow, so\uparrow; x\uparrow; [\neg li \wedge si]; lo\downarrow, [\neg ri]; ro\uparrow; x\downarrow; [ri]; ro\downarrow].$$

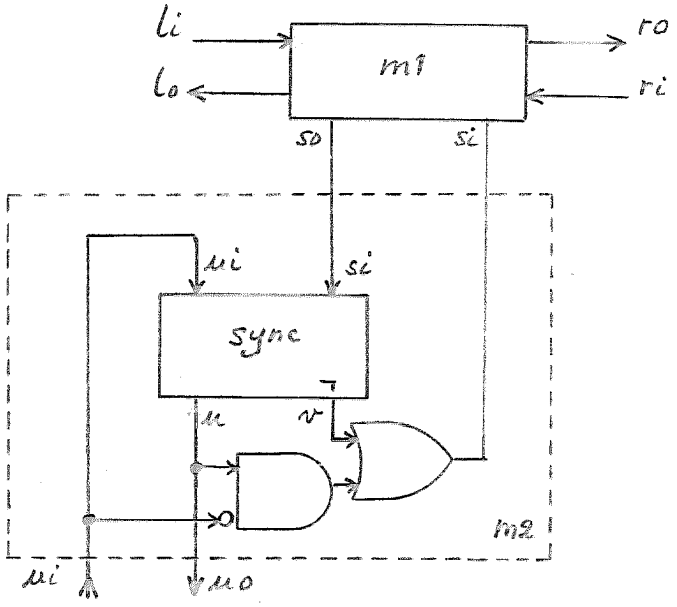


Figure 1: Circuit for m2

The production-rule expansion gives:

$$\begin{aligned} \neg si \wedge \neg ro \wedge li \wedge x &\mapsto so\uparrow, lo\uparrow \\ lo &\mapsto x\uparrow \\ si \wedge x \wedge \neg li &\mapsto lo\downarrow, so\downarrow \\ \neg lo \wedge x \wedge \neg ri &\mapsto ro\uparrow \\ ro &\mapsto x\downarrow \\ \neg x \wedge ri &\mapsto ro\downarrow \ . \end{aligned}$$

The special process that starts with $R$ is initialized simply by setting its variable $x$ to true.

We can improve the solution even further. We first observe that the use of the critical section takes place entirely between $so\uparrow$ and $[si]$ in $m1$. Hence, the action of passing the token to the right can start immediately after $[si]$. This gives the following reshuffling of the handshaking expansion:

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

The production-rule expansion does not require any state variable:

$$\begin{aligned} \neg si \wedge \neg ro \wedge li &\mapsto so\uparrow, lo\uparrow \\ lo \wedge \neg ri \wedge si &\mapsto ro\uparrow \\ ro \wedge \neg li &\mapsto lo\downarrow, so\downarrow \\ \neg lo \wedge ri &\mapsto ro\downarrow \ . \end{aligned}$$

The operator expansion gives the two generalized C-elements represented in Figure 2. The initialization of the process that starts holding the token is quite difficult with this PR expansion. A way out is to use the first implementation just for this process.

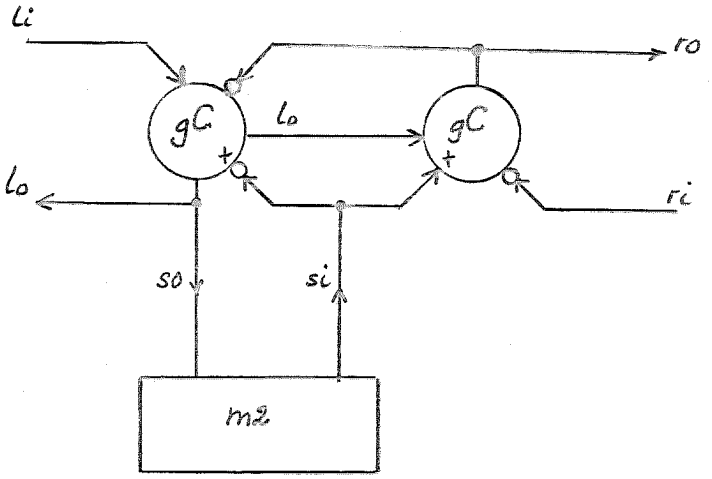


Figure 2: Four-phase implementation of m1

# 5 Two-phase implementation of m1

We can also implement $L$ and $R$ with two-phase handshake. Since $m1$ is a straight-line program, it is always known whether the handshake transi-

tions are upgoing or downgoing; and therefore this is a case where two-phase handshake can be implemented efficiently.

As usual, we unroll the loop once and get:

$$*[L; S; R; L; S; R] .$$

We choose to implement $L$ passive and $R$ active. But observe that $S$ has to be four-phase active because of the structure of the basic synchronizer. If we postpone the decision of whether $S$ should be lazy-active or not, we get:

$$m1 \equiv *[[li]; lo\uparrow; S; ro\uparrow; [ri]; [\neg li]; lo\downarrow; S; ro\downarrow; [\neg ri]] .$$

We can postpone $lo\uparrow$ until after $[ri]$ and $lo\downarrow$ until after $[\neg ri]$, and then decompose $m1$ into the two processes:

$$m11 \equiv *[[li]; S; ro\uparrow; [\neg li]; S; ro\downarrow]$$
$$m12 \equiv *[[ri]; lo\uparrow; [\neg ri]; lo\downarrow]$$

Process $m12$ is obviously a wire, and process $m11$ is a "two-to-four-phase converter," where $li$ and $ro$ are the handshake variables of the two-phase side, and $S$ is the four-phase side.

## 5.1 Phase Converters

The implementation of the converter is slightly different depending on whether $S$ is plain active or lazy active. The first case has already been implemented in [4]. The handshaking expansion with a state variable added gives:

$$\begin{array}{l} *[\,[li]; so\uparrow; [si]; u\uparrow; [u]; so\downarrow; [\neg si]; ro\uparrow; \\ \quad [\neg li]; so\uparrow; [si]; u\downarrow; [\neg u]; so\downarrow; [\neg si]; ro\downarrow \\ \,] . \end{array}$$

The rest of the compilation is left as an exercise for the reader. The circuit obtained consists of a toggle (constructed as two cross-coupled switches) and a difference element. It is shown in Figure 3.

For the case that $S$ is lazy active, the handshaking expansion with a state variable added gives:

$$\begin{array}{l} *[\,[li]; [\neg si]; u\uparrow; [u]; so\uparrow; [si]; so\downarrow; ro\uparrow; \\ \quad [\neg li]; [\neg si]; u\downarrow; [\neg u]; so\uparrow; [si]; so\downarrow; ro\downarrow \\ \,] . \end{array}$$

We replace $so\downarrow; ro\uparrow$ with $ro\uparrow; so\downarrow$. The production-rule expansion gives:

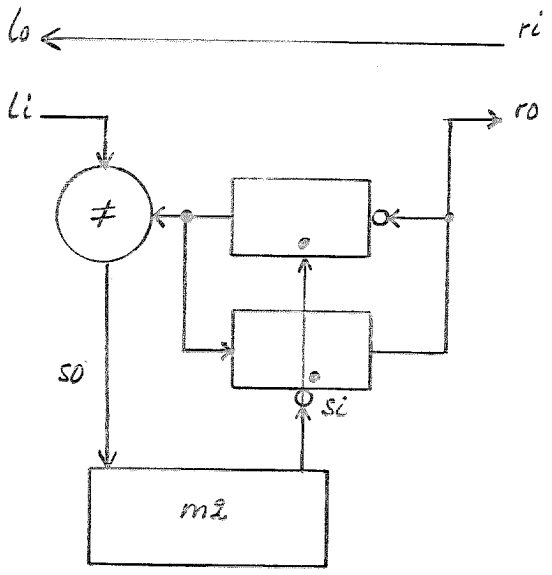


Figure 3: First two-phase implementation of m1

$$
\begin{aligned}
li \wedge \neg si &\mapsto u\uparrow \\
u \wedge \neg ro &\mapsto so\uparrow \\
si \wedge u &\mapsto ro\downarrow \\
ro \wedge u &\mapsto so\downarrow \\
\neg li \wedge \neg si &\mapsto u\downarrow \\
\neg u \wedge ro &\mapsto so\uparrow \\
si \wedge \neg u &\mapsto ro\downarrow \\
\neg ro \wedge \neg u &\mapsto so\downarrow \ .
\end{aligned}
$$

The operator reduction gives again two switches and a difference element, but connected in a different way than in the previous case. The circuit for $m1$ is shown in Figure 4.

In both cases, the initialization of the special process consists of just an inverter on the wire $ri \ \underline{w} \ lo$.

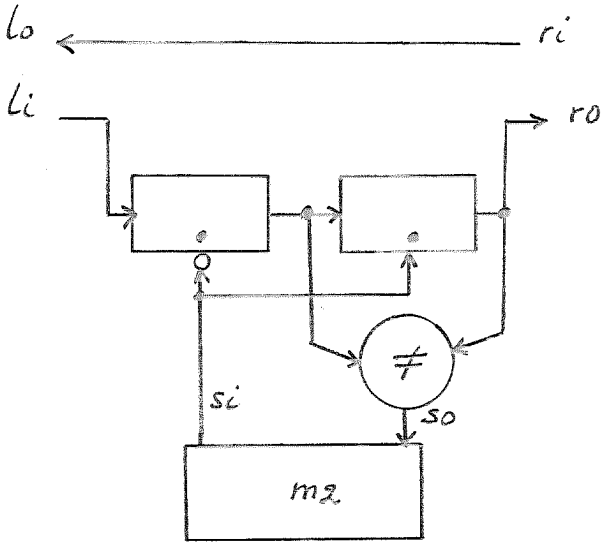


Figure 4: Second two-phase implementation of m1

## 6 Comparison of the Circuits

We shall compare the different solutions based on the number of transitions in series required for a server to pass the token from its left neighbor to its right neighbor. The four-phase solution requires the following sequence of firings:

$$gen.C, m2, gen.C$$

The first two-phase solution requires:

$$diff, m2, switch, diff, m2, switch$$

The second two-phase solution requires:

$$switch, switch, diff, m2, switch$$

(Actually, this implementation can be slightly improved by having the transitions on $u$ after the transitions on $so$. But the operators are not standard

and therefore a little less convenient from the point of view of the description of the circuit. Since the difference is marginal, we leave the other implementation as an exercise to the reader.)

Hence, the four-phase implementation is the most efficient, followed by the second two-phase implementation.

## Acknowledgments

Dražen Borković, Steve Burns, Pieter Hazewindus, and José Tierno contributed to the design of the different solutions.

## References

[1] Alain J. Martin. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming*, 5, 265-276, 1985.

[2] Alain J. Martin. The Design of a Self-timed Circuit for Distributed Mutual Exclusion. *1985 Chapel Hill Conference on VLSI*, ed. Henry Fuchs, Computer Science Press, 247-260, 1985.

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

[4] Alain J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. *UT Year of Programming Institute on Formal Developments of Programs and Proofs*, ed. E.W. Dijkstra, Addison-Wesley, Reading MA, 1989.

[5] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits. *UT Year of Programming Institute on Concurrent Programming*, ed. C.A.R. Hoare, Addison-Wesley, Reading MA, 1990.

[6] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. *Formal Methods for VLSI Design*, ed. J. Staunstrup, North-Holland, 1990.

**Note added in proof** A four-phase solution can easily be derived without using process decomposition. The circuit obtained is slightly more efficient than the one described above, but it is larger since each alternative path (depending whether $\overline{U}$ holds or not) requires its own control part.