
# Compiling Communicating Processes into Delay-Insensitive VLSI Circuits

Alain J. Martin

Computer Science Department
California Institute of Technology

5210:TR:86

# Compiling Communicating Processes into Delay-Insensitive VLSI Circuits

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA
5210:TR:86

*31 December 1985*
published in: *Distributed Computing*, (1986) 1:226-234

## 1. Introduction

If VLSI is an adequate technology to implement highly concurrent computations [7], it should be possible to apply to VLSI the already well-established design methods for distributed programming. Ideally, a distributed computation should be described in a notation that can be compiled into a VLSI-circuit as well into code for a stored-program computer. The method described in this paper is a step in that direction. At the moment, the term "compiling" means a "systematic, semantics-preserving transformation". The ultimate goal of the transformation being carried out automatically has not yet been achieved, although we believe that it is not remote.

In the method we propose, the computation is initially described as a set of communicating processes in the notation of [3], which is somewhat similar to C.A.R. Hoare's CSP [2]. This first description is the reference solution, which has to be proved correct. The program is then compiled into a delay-insensitive circuit by applying a series of semantics-preserving transformations. Hence the circuit obtained is correct by construction: all semantic properties that can be proved of the program hold for the circuit as well.

Following [11], a circuit is called *delay-insensitive* when its correct operation is independent of any assumption on delays in operators and wires, except that the delays are

finite. Consequently, such circuits do not use a clock signal: sequencing is enforced entirely by communication mechanisms. Delay-insensitive circuits have been known and used for their elegance, versatility, and robustness, which result from the ideal separation of concerns they provide between the mathematical and physical aspects of circuit design.

The first modern survey on the topic is [10], where such circuits are called *self-timed.* A different approach—the *macro-module* approach—is described in [8]. Closer to our method is the recent work at Eindhoven University of Technology, a good survey of which is [9].

A circuit is a network of elementary operators (*and*, *or*, *C*-element, arbiter, synchronizer, wire, fork). The specification of an operator is a so-called *production rule set*, where a production rule is a "weaker" form of guarded command, and a production rule set a "weaker" form of repetition. The compilation relies essentially on the four-phase (also called four-cycle) handshaking expansion of the communications. After expansion, the program of each process is compiled into a production rule set from which all explicit sequencing has been removed. By matching those production rules to those describing the operators, the programs are identified with networks of operators.

The method has already been applied to a whole spectrum of problems, some of them, such as distributed mutual exclusion [4], and fair arbitration [5], being quite difficult. The results are beyond our original expectations. For many circuits, especially complex ones, the compiled circuits are superior to their "hand-designed" counterparts, which are often more complex and not entirely delay-insensitive.

We first present the program notation and the VLSI operators that constitute the "object code". We then describe the four steps of the compilation and illustrate the method with a number of simple examples.

## 2. The program notation

### Sequential part

For the sequential part of the algorithm, we use a subset of Edsger W. Dijkstra's guarded command language [1], with a slightly different syntax. In this introductory paper we give only a very informal definition of the semantics of the constructs used.

i) $b\uparrow$ stands for $b := \mathbf{true}$, $b\downarrow$ stands for $b := \mathbf{false}$.

ii) The execution of the *selection* command $[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]$, where $G_1$ through $G_n$ are Boolean expressions, and $S_1$ through $S_n$ are program parts, ($G_i$ is called a "guard", and $G_i \rightarrow S_i$ a "guarded command") amounts to the execution of an arbitrary $S_i$ for which $G_i$ holds. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \ldots \vee G_n)$ holds. v

iii) For atomic actions $x$ and $y$, "$x, y$" stands for the execution of $x$ and $y$ in any order.

iv) $[G]$ where $G$ is a Boolean, stands for $[G \rightarrow \mathbf{skip}]$, and thus for "wait until $G$ holds". (Hence, "$[G]$; $S$"and $[G \rightarrow S]$ are equivalent.)

v) $*[S]$ stands for "repeat $S$ forever".

vi) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]]$ is "repeat forever: wait until some $G_i$ holds; execute an $S_i$ for which $G_i$ holds".

### Communicating processes

A concurrent computation is described as a set of processes composed by the usual parallel composition operator $\|$. Processes communicate with each other by communication actions on channel; they do not share variables. When no messages are transmitted, communication on a channel is reduced to synchronization signals. The name of the channel is then sufficient for identifying a communication action.

If two processes $p1$ and $p2$ share a channel named $X$ in $p1$ and $Y$ in $p2$, at any time the number of completed $X$-actions in $p1$ equals the number of completed $Y$-actions in $p2$. In other words, the completion of the $n$-th $X$-action "coincides" with the completion of the $n$-th $Y$-action. If, for example, $p1$ reaches the $n$-th $X$-action before $p2$ reaches the $n$-th $Y$-action, the completion of $X$ is suspended until $p2$ reaches $Y$. The $X$-action is then said to be *pending*. When thereafter $p2$ reaches $Y$, both $X$ and $Y$ are completed. The predicate "$X$ is pending" is denoted $\mathbf{q}X$. If, for an arbitrary command $A$, $\mathbf{c}A$ denotes the number of completed $A$-actions, the semantics of a pair $(X, Y)$ of communication commands is expressed by the two axioms:

$$\mathbf{c}X = \mathbf{c}Y \tag{A1}$$

$$\neg\mathbf{q}X \vee \neg\mathbf{q}Y. \tag{A2}$$

### Probe

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general Boolean command on channels, called the *probe*. The definition of the probe given in [3] states that in process $p1$, the probe command $\overline{X}$ has the same value as $\mathbf{q}Y$. Here, we use a weaker definition, namely:

$$\overline{X} \Rightarrow \mathbf{q}Y$$

$$\mathbf{q}Y \Rightarrow \diamond\overline{X},$$

where $\diamond P$ means $P$ *holds eventually*.

Hence the guarded command $\overline{X} \rightarrow X$ guarantees that the $X$-action is not suspended. And a construct of the form $[\overline{X} \rightarrow X \mid \overline{Y} \rightarrow Y]$ can be used for selection. (For a more rigorous definition of the communication mechanism and the probe, see [3].)

## 3. The "Object Code"

The set of operators with which we want to build our circuits is not unique. In this introduction, we will use the simple set consisting of *and*, *or*, *C-element*, *wire*, and *fork*. We believe that this simple set extended with an *arbiter* and a *synchronizer* is sufficient for compiling any program. Each operator is described by a set of production rules. A production rule is similar to a guarded command, and we shall therefore use a similar syntax. There are, however, important semantic differences. Consider the production rule $G \mapsto S$:

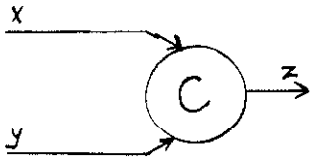
- $S$ is either a simple assignment or of the form "$s1, s2$" where $s1$ and $s2$ are each a simple assignment.

- If $G$ holds, the correct execution of $S$ is guaranteed only if $G$ remains invariantly **true** until the completion of $S$. We say that $G$ must be *stable*.
- Unlike the guarded commands of a selection or a repetition, the mutual exclusion among the different production rules of a set is not guaranteed automatically. It has to be enforced by the semantics of the program.
- If stability of the guards and mutual exclusion among guards are guaranteed, the production rule set $PRS$ is semantically equivalent to the repetition $*[[GCS]]$, where $GCS$ is the guarded command set syntactically identical to $PRS$.

The description of the five operators used in this paper in terms of their production rules and their logic symbols are as follows.

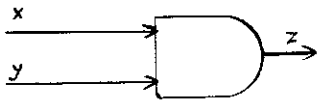
The C-element:

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



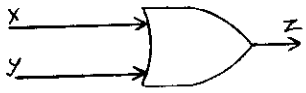

The "and":

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



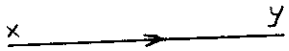

The "or":

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



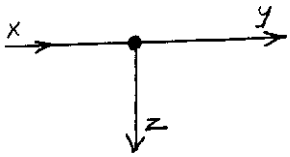

The wire:

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




The fork:

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




Any input or output variable of an operator may be negated. In particular, a wire with its input or its output negated—but not both—is an inverter. A negated input or output is represented in the figures by a small circle on the corresponding line.

## 4. The Compilation Method

### Process Decomposition

The first step of the compilation, called "process decomposition", consists in replacing a process by several semantically equivalent processes. The purpose of the decomposition is to obtain a process representation of the program in which the right-hand side of each

guarded command is a straight-line program, i.e. consists only of simple assignments and communication commands, composed by semi-colons and commas.

**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 action $C$ 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 $P1$, much as a procedure or a coroutine would be. The only purpose of this transformation is to simplify the structure of each command. As an example, consider the process:

$$P \equiv *[[...A; [B_1 \rightarrow S_1 \mid B_2 \rightarrow S_2]; ...]].$$

Applying the decomposition rule, $P$ is replaced by the two processes $P1$ and $P2$. Channel $(C, D)$ is introduced between $P1$ and $P2$.

$$\begin{aligned} P1 &\equiv *[[...A; C; ...]] \\ P2 &\equiv *[[\overline{D} \wedge B_1 \rightarrow S_1; D \\ &\qquad \mid \overline{D} \wedge B_2 \rightarrow S_2; D \\ &\qquad ]]. \end{aligned}$$

Observe that the newly created processes $P1$ and $P2$ may share variables. Since the processes are never active concurrently, there is no conflicting access to the shared variables. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program.

### Handshaking Expansion

The implementation of communication, called "handshaking expansion", replaces each channel by a pair of wire-operators and each communication action by its implementation. Channel $(X, Y)$ is implemented by the two wires $(xo \ \underline{w} \ yi)$ and $(yo \ \underline{w} \ xi)$.

If $X$ belongs to process $p1$ and $Y$ to process $p2$, $xo$ and $xi$ belong to $p1$, and $yo$ and $yi$ belong to $p2$. Initially, $xo$, $xi$, $yo$, and $yi$—which we will call the "handshaking variables of $(X, Y)$"—are **false**. Assume that the program has been proved to be deadlock-free and that we can identify a pair of matching actions $X$ and $Y$ in $p1$ and $p2$ respectively. We replace $X$ and $Y$ by the sequences $U_x$ and $U_y$ respectively, with:

$$\begin{aligned} U_x &\equiv xo\uparrow;\ [xi] \\ U_y &\equiv [y_i];\ yo\uparrow. \end{aligned}$$

The formal proof that $U_x$ and $U_y$ fulfil axioms $A1$ and $A2$ is omitted. The following is an informal argument that relies on a definition of completion of an action different from the usual one. Since the argument is not essential to the comprehension of the method, it may be skipped at first reading.

Assume that we know what the *initiation* and *termination* of an atomic action mean. A non-atomic action is initiated when its first atomic action is initiated. A non-atomic action is terminated when its last atomic action is terminated.

*A non-atomic action is said to be* **completed** *when it is initiated and it is guaranteed to terminate.*

(An atomic action is completed when it is terminated.) Between initiation and completion, an action is *suspended.*

Obviously, $U_x$ and $U_y$ are guaranteed to terminate if and only if they are both initiated, which establishes $A1$ and $A2$.

It is essential to observe that these definitions of completion and suspension are valid because they satisfy the semantic properties of completion and suspension that are used in correctness arguments, namely:

$$\{\mathbf{c}X = x\}\ X\ \{\mathbf{c}X = x + 1\}$$

$$\mathbf{q}X \Rightarrow pre(X)$$

where $pre(X)$ is any precondition of $X$ in terms of the program variables and auxiliary program variables.

(This completes the argument.)

Unfortunately, when the communication terminates, all handshaking variables are **true**. Hence, we cannot implement the next communication with $U_x$ and $U_y$. However, the complementary implementation can be used for the next matching pair, namely:

$$D_x \equiv xo\downarrow;\ [\neg xi]$$
$$D_y \equiv [\neg yi];\ yo\downarrow .$$

The solution consisting in alternating $U_x$ and $D_x$ as an implementation of $X$, and $U_y$ and $D_y$ as an implementation of $Y$ is essentially the so-called "two-phase handshaking", or "two-cycle signaling". However, it is in general not possible to determine syntactically which $X$- or $Y$-actions are following each other in an execution. In general, two-phase handshaking implementations require testing the current value of the variables. In this paper, we shall use a simpler but less efficient solution known as "four-phase handshaking", or "four-cycle signaling".

In a four-phase handshaking protocol, all $X$-actions are implemented as "$U_x; D_x$" and all $Y$-actions as "$U_y; D_y$". Observe that the $D$-parts in $X$ and $Y$ introduce an extra communication between the two processes whose only purpose is to reset all variables to **false**. The synchronization introduced by this extra communication is unnoticeable since the immediately preceding communication implemented by $U_x$ and $U_y$ sees to it that both processes reach a matching $D_x$ and $D_y$ "at the same time".

Both protocols have the property that for a matching pair $(X, Y)$ of actions, the implementation is not symmetrical in $X$ and $Y$. One action is called *active* and the other one *passive*. The four-phase implementation with $X$ active and $Y$ passive is:

$$X \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \qquad (1)$$

$$Y \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow \tag{2}$$

When no action of a matching pair is probed, the choice of which one should be active and which one passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that for a given channel $(X, Y)$, all actions at one side are active and all actions at the other side passive. If $\overline{X}$ is used, all $X$-actions are passive—with the obvious restriction that $\overline{Y}$ cannot be used in the same program.

The implementation of the probe is simply:

$$\begin{aligned} \overline{X} &\equiv xi \\ \overline{Y} &\equiv yi \end{aligned} \tag{3}$$

Given our definition of suspension, the proof that this implementation of the probe fulfils the definition of Section 2 is straightforward and is omitted.

A probed communication action $\overline{X} \rightarrow \ldots X$ is implemented:

$$xi \rightarrow \ldots xo\uparrow;\ [\neg xi];\ xo\downarrow .$$

### Basic properties

The following properties of the handshaking protocol play an important role in the compilation method.

**Property 1:** *For the pair of wires $(xo \underline{w}\, yi)$ and $(yo \underline{w}\, xi)$, used together as in (1) and (2), and all variables* **false** *initially, the following sequence of transitions is guaranteed to occur if the system is deadlock-free:*

$$*[xo\uparrow;\ yi\uparrow;\ yo\uparrow;\ xi\uparrow;\ xo\downarrow;\ yi\downarrow;\ yo\downarrow;\ xi\downarrow]. \tag{4}$$

*Hence, the following postconditions hold:*

$$\begin{aligned} &xo\uparrow\{\diamond xi\} \\ &xo\downarrow\{\diamond\neg xi\} \\ &yo\uparrow\{\diamond\neg yi\} \end{aligned} \tag{5}$$

**Property 2:** *Consider the handshaking expansion of a program $p$ according to (1), (2), and (3). Provided that the cyclic order of the four handshaking actions of a communication command is respected, the last two actions of this command—the two actions of $D_x$ or $D_y$—can be inserted at any place in $p$ without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of $p$ may introduce deadlock.*

Property 2 is a direct consequence of the way in which we have introduced the sequences $D_x$ and $D_y$. We will see examples of how to use Property 2. In this paper, we will ignore the deadlock issue when we re-order handshaking actions.

### First example: stack element

Consider the simple process $S$, which we call a"stack element":

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

where $L$ and $R$ are channels. Since $L$ is probed, it must be passive, and if we want to compose $S$-processes together, $R$ must be active, since it will match a passive $L$. The handshaking expansion gives:

$$*[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ lo\downarrow]. \tag{6}$$

## 5. Production-rule expansion

The next step is to compile the handshaking expansion of the program into a set of production rules from which all explicit sequencing has been removed. By matching those production rules to those describing the semantics of operators, the programs can be identified with networks of operators. We use the compilation of $S$ to illustrate the different steps of the expansion.

We start with the production rule set syntactically derived from the program. In the case of $S$, it is the set derived from (6), namely:

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

The execution of a production rule is called *effective* if it changes the value of a variable. Otherwise, it is called *vacuous*. We ignore vacuous executions of production rules.

For each guarded command of the program, the production rule set representation is semantically equivalent to the program representation if and only if the order of execution of effective production rules is the same as the order of the corresponding transitions in the program—we call it the *program order*. (As a clue to the reader we list the production rules of a set in program order.)

In general, we have to strengthen the guards of some rules to enforce execution in program order. This is the case in our example: Since $\neg ri$ holds initially, the third production rule can be executed first. It is also true for the fourth production rule; but the execution of the fourth rule in the initial state is vacuous.

Because all handshaking variables of $R$ are back to **false** when $R$ is completed, we cannot find a guard for the transition $lo\uparrow$. (Hence, the transitions following a semi-colon that can be identified with a semi-colon of the original program are likely to be difficult to deal with.)

### Direct implementation

In order to define uniquely the state in which the transition $lo\uparrow$ is to take place, the first technique consists in introducing a state variable, say $x$, initially **false**. $S$ becomes

$$*[[li];\ ro\uparrow;\ [ri];\ x\uparrow;\ [x];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ x\downarrow;\ [\neg x];\ lo\downarrow]. \tag{7}$$

Now, the production-rule expansion can be performed:

$$\begin{aligned}
\neg x \wedge li &\mapsto ro\uparrow\{\diamond ri\} && (S1)\\
ri &\mapsto x\uparrow\{x\} && (S2)\\
x &\mapsto ro\downarrow\{x \wedge \diamond\neg ri\} && (S3)\\
x \wedge \neg ri &\mapsto lo\uparrow\{\diamond\neg li\} && (S4)\\
\neg li &\mapsto x\downarrow\{\neg x\} && (S5)\\
\neg x &\mapsto lo\downarrow. && (S6)
\end{aligned}$$

(Why is the conjunct $\neg x$ necessary in the first rule?) Using the postconditions indicated between braces—these conditions rely on (5)—, it is easy to verify that the production rules of the set are executed in program order. Hence, the execution of the production rule set is equivalent to the execution of (7).

### Re-ordering implementation

Another way to find a valid guard for $lo\uparrow$ is to use Property 2, to re-order the actions of (6). For instance, we can postpone the second half of the handshaking expansion of $S$ —i.e., the sequence $ro\downarrow;[\neg ri]$—until after $[\neg li]$. We get:

$$*[[li];\ ro\uparrow;\ [ri];\ lo\uparrow;\ [\neg li];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]. \tag{8}$$

The syntactic production rule expansion is already "program ordered":

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

## 6. Operator reduction

The last step of the compilation, called *operator reduction*, consists in identifying sets of production rules in the program with sets of production rules describing operators. The program can then be identified with a set of operators. We group pairs of production rules that modify the same variable.

If a given group cannot be directly identified with the production rule set of an operator, we perform on this group a last transformation called *symmetrization*: we transform the guards of the production rules—again under invariance of the semantics—so as to make them "look like" the guards of operators. In case a guard contains too many variables, this step may also involve decomposing a production rule into several production rules by introducing new internal variables.

Consider $S1$ and $S3$. No operator corresponds to these rules. But, if we replace $x$ by $\neg li \vee x$ in $S3$, the value of the guard of $S3$ is not changed since $li$ holds as precondition

of $S3$, and now the two production rules represent the operator $(\neg x, li) \triangle ro$ . Since we have weakened the guard of $S3$, we have to check that we have not enlarged the set of states in which $S3$ can be effectively executed. No such state has been added, hence the transformation is safe.

In the case of $S2$ and $S5$, no guard can be weakened. We therefore strengthen both of them as

$$ri \wedge li \mapsto x\uparrow$$
$$\neg ri \wedge \neg li \mapsto x\downarrow,$$

which corresponds to the $C$-element $(ri, li) \underline{C}\, x$. Observe that strengthening the guards in this way is always possible since the guards are mutually exclusive by construction. Hence it is always possible to implement a pair of guards with a $C$-element. Why then bother about weakening the guards? The answer is that introducing a disjunction is the only transformation leading to combinatorial operators—*and, or*—, which are usually less "expensive" than $C$-elements—a $C$-element is a state-holding operator.

For the direct implementation of $S$, the symmetrization of the set $S1$ through $S6$ gives:

$$\neg x \wedge li \mapsto ro\uparrow \qquad (S1)$$
$$ri \wedge li \mapsto x\uparrow \qquad (S2)$$
$$\neg li \vee x \mapsto ro\downarrow \qquad (S3)$$
$$x \wedge \neg ri \mapsto lo\uparrow \qquad (S4)$$
$$\neg ri \wedge \neg li \mapsto x\downarrow \qquad (S5)$$
$$ri \vee \neg x \mapsto lo\downarrow. \qquad (S6)$$

The identification with operators is now straightforward.

($S1$, $S3$) corresponds to $(\neg x, li) \triangle ro$.

($S2$, $S5$) corresponds to $(li, ri)\, \mathbf{C}\, x$.

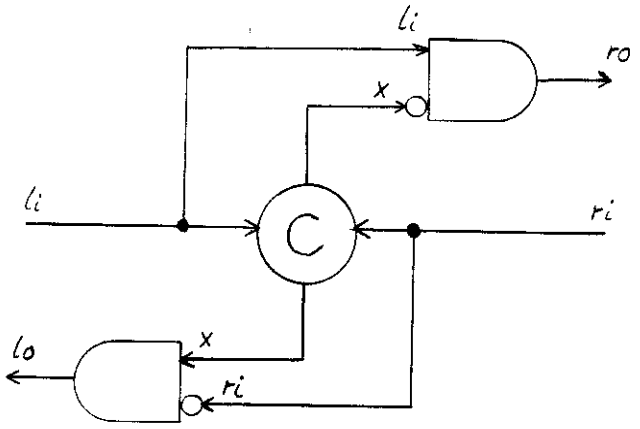
($S4$, $S6$) corresponds to $(x, \neg ri) \triangle lo$.

### Isochronic forks

In the previous operator reduction, $li$ is input to the $C$-element $(li, ri) \underline{C}\, x$, and to the *and*-operator $(li, \neg x) \triangle ro$. Formally, in order to compose the circuit we have to introduce the fork $li\, \underline{f}\, (l1, l2)$ and replace $li$ by $l1$ in the $C$-element and by $l2$ in the *and*-operator.

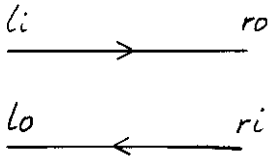
Since the fork is delay-insensitive, $l1$ and $l2$ are not guaranteed to have the same value in all states, whereas the two operators are constructed with the same input variable $li$. We solve this problem by making a simplifying assumption: we assume that the forks used to connect operators inside a process are *isochronic*, i.e. the delays in these forks are short enough, compared to the delays in all operators other than forks and wires, to assume that the two outputs of an isochronic fork have the same value at any time.

The resulting circuit is shown in Fig. 1.




–Figure 1–

For the second implementation of $S$—with re-ordering of actions—the production rule set can be reduced directly: the first and third rules specify the wire $li \underline{w} ro$, the second and fourth rules specify the wire $ri \underline{w} lo$. The circuit is shown in Fig. 2.




–Figure 2–

Comparing the circuits of Figs. 1 and 2, we observe that the re-ordering of handshaking actions leads to a simpler implementation. This observation is true in general, although the gain is not always as drastic as in this case. We also observe that re-ordering handshaking actions modifies the behavior of the circuit concerning its synchronization with its environment. This is not surprising since the second half of a handshaking sequence—the part that we shift from its place—is an extra synchronization action. Placed just after the first half, this second synchronization has no noticeable effect. But its synchronization effect becomes noticeable when the action is shifted away from the first half of the handshaking sequence. Hence the choice to re-order actions is a choice in favor of a simpler circuit at the cost of modifying the original synchronization behavior of the circuit—in general for the worse.

## 7. Second example: one-place buffer

Our second example is the simple "one-place buffer" process

$$B \equiv *[L;\ R],$$

where $L$ and $R$ are two channels. The handshaking expansion of $B$ gives:

$$B \equiv *[[li];\ lo\uparrow;\ [\neg li];\ lo\downarrow;\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri]]. \tag{9}$$

Here the difficult transition is $ro\uparrow$. In this example we construct only the solution obtained by re-ordering of actions. The construction of the solution with introduction of a state variable is more difficult and is left as an exercise to the reader. (It is described in [6].) If we postpone the second half of the handshaking expansion of $L$ until after $[ri]$, we get:

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

which we can also re-order as:

$$*[[\neg ri];\ [li];\ lo\uparrow;\ ro\uparrow;\ [ri];\ [\neg li];\ lo\downarrow;\ ro\downarrow]. \tag{10}$$

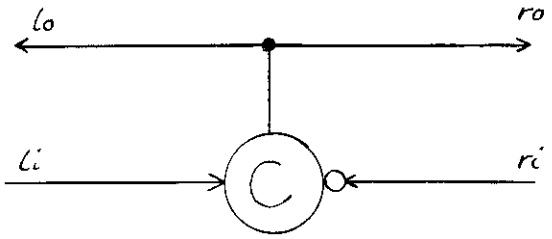
The order between two successive transitions on output variables—like $lo\uparrow;\ ro\uparrow$—is irrelevant. Hence the production-rule expansion of (10) gives:

$$\neg ri \wedge li \rightarrow lo\uparrow, ro\uparrow$$
$$ri \wedge \neg li \rightarrow lo\downarrow, ro\downarrow .$$

After introducing the auxiliary variable $u$, the production rule expansion is straightforward:

$$((\neg ri, li)\ \underline{C}\ u)$$
$$(u\ \underline{f}\ (lo, ro)).$$

The corresponding circuit is shown in Fig. 3.




–Figure 3–

## 8. Message communication

So far, we have only considered the synchronization aspect of the communication actions: no message was passed. The last two examples describe implementations of communications that entail transmissions of messages. We consider the transmission of Boolean variables only; the generalization to other types is relatively straightforward.

### Third example: Queue (FIFO) element

Queues (FIFO) play an important role in pipeline computations for increasing throughput when processing times are variable. A queue consists of the linear composition of a number of buffer-elements of the type:

$$E \equiv *[L?(x); R!(x)]\,. \tag{11}$$

($L?(x)$ is an input action assigning to internal variable $x$ the value received on $L$. $R!(x)$ is an output sending the value of $x$ on channel $R$.)

We are going to implement the transmission of **true** messages and of **false** messages on two independent channels. We shall construct a circuit for each type of messages, and then compose the two circuits. Such a technique is called the "double-rail" technique [10]. We get:

$$\begin{array}{l} *[[\overline{L}_t \rightarrow L_t;\ R_t \\ \ \ |\overline{L}_f \rightarrow L_f;\ R_f \\ \ \ ]], \end{array}$$

where $\neg\overline{L}_t \vee \neg\overline{L}_f$ holds at any time.

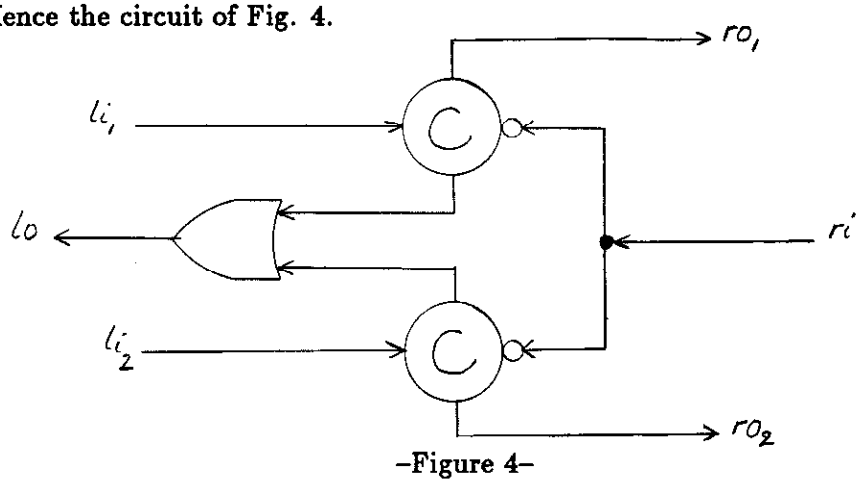
If we let channels $L_t$ and $L_f$ share variable $lo$, and channels $R_t$ and $R_f$ share variable $ri$, the handshaking expansion gives the two guarded commands:

$$\begin{array}{l} *[[li_1 \rightarrow lo\uparrow;\ [\neg li_1];\ lo\downarrow;\ ro_1\uparrow;\ [ri];\ ro_1\downarrow;\ [\neg ri] \\ \ \ |li_2 \rightarrow lo\uparrow;\ [\neg li_2];\ lo\downarrow;\ ro_2\uparrow;\ [ri];\ ro_2\downarrow;\ [\neg ri] \\ \ \ ]]. \end{array} \tag{12}$$

The production rule expansion of (12) has to guarantee mutual exclusion between the two guarded commands. Since $\neg li_1 \vee \neg li_2$ holds at any time, it is easy to see that mutual exclusion is guaranteed if we re-order the actions of each guarded command as in the implementation of $B$. We get:

$$\begin{array}{l} *[[\neg ri \wedge li_1 \rightarrow lo\uparrow, ro_1\uparrow;\ [ri \wedge \neg li_1];\ lo\downarrow, ro_1\downarrow \\ \ \ |\neg ri \wedge li_2 \rightarrow lo\uparrow, ro_2\uparrow;\ [ri \wedge \neg li_2];\ lo\downarrow, ro_2\downarrow \\ \ \ ]]. \end{array} \tag{13}$$

Since each of the two guarded commands of (13) is identical to (10), the circuit for (12) consists of two copies of the circuit of Fig. 3 composed in the obvious way so as to share $lo$ and $ri$. Hence the circuit of Fig. 4.




–Figure 4–

### Fourth example: single variable

Consider the following process that provides read and write access to a simple Boolean variable $x$:

$$\begin{array}{l} *[[\overline{P} \rightarrow P?x \\ \quad |\overline{Q} \rightarrow Q!x \\ \quad ]], \end{array} \tag{14}$$

where $\neg\overline{P} \vee \neg\overline{Q}$ holds at any time.

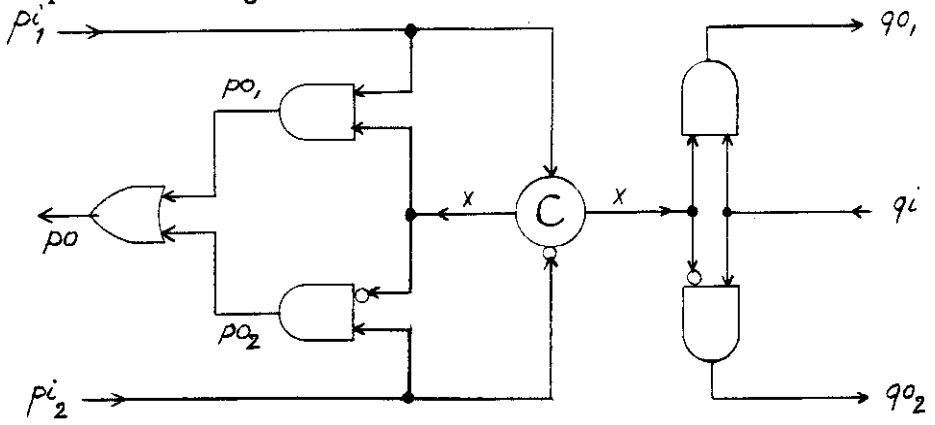
Again, according to the double-rail technique, each guarded command of (14) is expanded to two guarded commands. But now the values **true** and **false** have to be explicitely assigned to $x$, in the following way:

$$\begin{array}{l} *[[pi_1 \rightarrow x\uparrow;\ [x];\ po\uparrow;\ [\neg pi_1];\ po\downarrow \\ \quad |pi_2 \rightarrow x\downarrow;\ [\neg x];\ po\uparrow;\ [\neg pi_2];\ po\downarrow \\ \quad |x \wedge qi \rightarrow qo_1\uparrow;\ [\neg qi];\ qo_1\downarrow \\ \quad |\neg x \wedge qi \rightarrow qo_2\uparrow;\ [\neg qi];\ qo_2\downarrow \\ \quad ]]. \end{array} \tag{15}$$

The rest of the compilation is now straightforward and is left as an exercise to the reader. (Hint: don't forget to ensure mutual exclusion between the guarded commands.) The operator reduction gives:

$$\begin{array}{l} (pi_1, \neg pi_2)\ \underline{C}\ x \\ (pi_1, x)\ \triangle\ po_1 \\ (pi_2, \neg x)\ \triangle\ po_2 \\ (po_1, po_2)\ \underline{\vee}\ po \\ (x, qi)\ \triangle\ qo_1 \\ (\neg x, qi)\ \triangle\ qo_2. \end{array}$$

The circuit is represented in Fig.6.




–Figure 6–

## 9. Conclusion

We have described a method for implementing a high-level concurrent algorithm (a set of communicating processes) as a network of digital operators that can be directly mapped into a delay-insensitive VLSI-circuit. The circuit is derived from the program by a series of systematic, semantics-preserving, transformations that we have compared to compiling.

Since the circuits are correct by construction, and in particular, since the guards of the production rules are stable by construction, the circuits are free from "hazards".

The choice between active and passive implementations is usually clear from the context. For instance, the choice to implement input as passive and output as active is most of the time safe. Furthermore, in the case the wrong choice has been made and it turns out that two active or two passive commands have to be paired, an "adaptor" process can be used. An adaptor is a one-place buffer with $L$ and $R$ both active—a "double-A"—or both passive—a "double-P". A double-A is used to pair two passive commands, a double-P to pair two active commands.

The simplifying assumption of isochronic forks is not severe, since such a fork is always confined to a very small circuit part. In fact, it is even weaker than the usual isochronic assumption used in self-timed design, where a whole circuit part is assumed isochronic. We believe that isochronic forks can be avoided, but doing so would complicate the circuits without real advantage in return.

We also believe that the basic sets of operators used in this paper, extended with an arbiter and a synchronizer to implement mutual exclusion among independent commands, is sufficient for all purposes. (Obviously, having both *and* and *or* is redundant.) However, there is no interest in confining the designer to a minimal set of operators. On the contrary, since one of the advantages of VLSI is the possibility to create operators at no cost, introducing other operators—like, e.g., *and* and *or* with more than two inputs, or *exclusive-or*—may often simplify a circuit drastically.

We have illustrated the method with four simple—sometimes deceivingly so—but characteristic examples that embody very standard control and data structures. The method has also been tested on quite difficult examples like the distributed mutual exclusion circuit described in [4]. In [5], we have used the method to solve an open problem: It had been conjectured that it is impossible to construct a delay-insensitive fair arbiter. We have disproved the conjecture by constructing such an arbiter applying our method.

The most encouraging aspect of the method is that it is really a synthesis technique: it allows a designer to construct solutions that he would never have found had he not applied the method.

## 10. Acknowledgement

I am indebted to Martin Rem, Chuck Seitz, Peggy Li, and Kevin Van Horn for their comments on the manuscript. Kevin Van Horn also contributed to the definition of the completion of a communication. Acknowledgements are also due to the Eindhoven VLSI Club, in particular Huub Schols, for their comments and criticisms during an oral presentation of this material in September 1985.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-79-C-0597.

## 11. References

[1] Dijkstra, Edsger W., *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ (1976)

[2] Hoare, C.A.R. "Communicating Sequential Processes". *Comm. ACM* 21,8, pp 666-677 (August 1978)

[3] Martin, A.J., "The Probe: an Addition to Communication Primitives", *Information Processing letters* 20, pp 125-130 (1985)

[4] 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 (1985)

[5] Martin, A.J., "A Delay-Insensitive Fair Arbiter", Caltech Computer Science Technical Report 5193:TR:85 (1985)

[6] Martin, A.J., "FIFO: an Exercise in Compiling Programs into Circuits", Caltech Computer Science Technical Memo (1985)

[7] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)

[8] Molnar, C.E., et al. "Synthesis of Delay-Insensitive Modules", *Proc. 1985 Chapel Hill Conf, VLSI*, ed. Henry Fuchs, pp 67-86, (1985)

[9] Rem, M., "Concurrent Computations and VLSI Circuits", in "Control Flow and Data Flow: Concepts in Distributed Programs", ed. M.Broy, pp 399-437 Springer-Verlag Berlin Heidelberg (1985).

[10] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)

[11] Snepscheut, J.v.d., "Trace Theory and VLSI Design" LNCS 200, Springer-Verlag Berlin Heidelberg (1985).