A Synthesis Method for Self-timed VLSI Circuits

Alain J. Martin

Department of Computer Science
California Institute of Technology

5256:TR:87
**A Synthesis Method for Self-Timed VLSI Circuits**

**Defense Advanced Research Projects Agency,** 3701 North Fairfax Drive, Arlington, VA 22203-1714

Approved for public release; distribution unlimited

14. ABSTRACT

see report

15. SUBJECT TERMS

unclassified

16. SECURITY CLASSIFICATION OF:

a. REPORT
b. ABSTRACT
c. THIS PAGE

17. LIMITATION OF ABSTRACT

18. NUMBER OF PAGES

19a. NAME OF RESPONSIBLE PERSON

unclassified

8

unclassified

Standard Form 298 (Rev. 8-98)

Prescribed by ANSI Std Z39-18
A Synthesis Method for Self-Timed VLSI Circuits

Alain J. Martin

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order No. 6202, and monitored by the office of Naval Research under contract number N00014-87-K-0745

©California Institute of Technology, 1987

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

5256:TR:87

Published in: Proc. 1987 IEEE International Conference on Design: VLSI in Computers & Processors
ICCD '87, Rye Town Hilton, Rye Brook, New York
October 5 - October 8, 1987; pp 224-229
IEEE Computer Society Press
A Synthesis Method for Self-timed VLSI Circuits

Alain J. Martin

Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

1. Introduction
With chip size reaching 1 million transistors, the need for high-level design of circuits becomes compelling. The main stumbling block in the development of design methods for VLSI algorithms is to find an interface that provides a good separation of the physical and algorithmic concerns. Among the physical issues, timing is the most critical, since it is not only essential to the real-time behavior of a circuit, but also to its logical correctness if synchronous techniques are used.

Synchronous techniques are detrimental to the use of high-level design methods because they don't "scale well": a circuit may cease to function correctly when its feature sizes are scaled down to smaller dimensions. Further, with the increasing size of circuits, it becomes more and more difficult to distribute safely a clock signal across a chip, and the restrictions attached to wire lengths in order to maintain certain timing properties add extra complication to the already difficult layout problem.

For all those reasons, self-timed techniques (as defined in [10]) are particularly attractive for high-level VLSI design [9]. We propose a synthesis method for self-timed circuits in which the computation is initially described as a set of communicating processes in the notation of [9], which is similar to C.A.R. Hoare's CSP [2] but augmented with the probe construct. This first description is the reference solution, which has to be proved correct. The program is then compiled into a self-timed circuit by applying a series of semantics-preserving transformations. Hence the circuit obtained is correct by construction.

Unlike most silicon compilation methods and hardware description languages, the method leads to efficient circuits. It has been applied with "hand compilation" to a series of difficult self-timed design problems, such as distributed mutual exclusion, fair arbitration, routing automata, with great success. Actually, the method, applied by a person in a mechanical way, will typically produce better results than the most experienced designers can produce. The main reason for the efficiency of the method is that, rather than going in one step from the program notation to the circuit, the designer applies a series of transformations to the original program. At each level of the transformation, powerful algebraic manipulations can be performed leading to important optimizations in terms of speed or area.

We shall 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 one sizeable example, the construction of a stack. We shall conclude that this technique can be used for high quality and high complexity designs, fully automated from a provably correct high-level description. (For a more complete description of the method, see [4], [5], [6], and [7].)

2. The program notation
The language used for the high-level description is close to C.A.R. Hoare's CSP[2]. We give only a very informal definition of the constructs used in this paper.

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

ii) The execution of the selection command (generalized IF-statement) $[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n]$, where $G_i$ 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 $\neg(G_1 \lor \ldots \lor G_n)$ holds, the execution of the command is suspended until $(G_1 \lor \ldots \lor G_n)$ holds.

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 $[G \rightarrow \text{skip}]$, 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 $a([G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n])$ is "repeat forever: wait until some $G_i$ holds; execute $S_i$ for which $G_i$ holds".

Communicating processes
A concurrent computation is described as a set of processes communicating with each other by communication actions on channels (no shared variables). When no messages are transmitted, communication on a channel is reduced to synchronisation signals. The name of the channel is then sufficient for identifying a communication action.

If two processes $p_1$ and $p_2$ share a channel named $X$ in $p_1$ and $Y$ in $p_2$, at any time the completion of the nth $X$-action "coincides" with the completion of the nth $Y$-action. If, for example, $p_1$ reaches the nth $X$-action before $p_2$ reaches the nth $Y$-action, the completion of $X$ is suspended until $p_2$ reaches $Y$. The $X$-action is then said to be pending.

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. In process $p_1$, the probe command $\overline{X} \rightarrow X$ guarantees that the $X$-action is not suspended. And a construct of the form $[\overline{X} \rightarrow X | \overline{Y} \rightarrow Y]$ can be used for selection.

3. The "object code"
In standard digital VLSI design, the MOS transistor is idealized as an on/off switch. Unfortunately, the switch model is too crude, ignoring too many electrical phenomena that play
an important role in the functioning of the circuit. Therefore, trying to carry the discrete model of a computation down to the transistor level is very likely to lead either to incorrect implementations or to a too complicated model of the computation. A crucial decision in the development of our method has been to choose an "object code" at a higher level than the transistor. We have chosen to construct a notation that is guided by the weakest possible form of control structure and relatively few of program constructs. In fact, the notation contains exactly one construct, the production rule, and is therefore called the "production-rule set notation".

This minimal notation has been chosen so that i) it has sound semantics, ii) any non-terminating program can be compiled into production rules, and the transformation into a circuit is straightforward.

In fact, we consider the production-rule set as the canonical representation of a circuit. This representation can be decomposed into several equivalent networks of gates depending on the set of building blocks used, but the production-rule set represents the circuit independently of the gate implementations.

4. Production rules
Production rules can be seen as a weaker form of guarded commands. Consider the production rule $G \rightarrow S$

- $S$ is either a simple assignment, or an unordered list of simple assignments, or a single assignment, or a single assignment of true or false to a single Boolean variable.

- $G$ is a Boolean expression, called the guard of the production rule. 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.

A production rule set is an unordered set (a collection) of production rules. Consider the canonical production rule set $PRS$

$$\begin{align*}
G_1 & \rightarrow S_1 \\
G_2 & \rightarrow S_2 \\
& \vdots \\
G_n & \rightarrow S_n
\end{align*}$$

- Unlike the guarded commands of a selection or a repetition, the mutual exclusion among the different production rules of a set is not part of the semantics of the construct. The correct execution of a production rule set is guaranteed only if interfering production rules are mutually exclusive. Two production rules are said to be interfering when their right-hand sides share a variable. Each process will be implemented as a p.r.s. such that exactly one p.r. is firing at any time, hence enforcing non-interference.

- If stability of the guards and mutual exclusion among interfering production rules are guaranteed, the production rule set $PRS$ is semantically equivalent to the non-terminating repetition $\forall[GCS]$, where $GCS$ is the guarded command set syntactically identical to $PRS$. Stability of the guards is essential to guarantee the absence of races and hazards. When stability cannot be enforced, a special operator called "synchronizer" has to be used. When mutual exclusion cannot be enforced, a special operator called "arbiter" has to be used. These two operators are not needed in this paper.

We implement a p.r.s. by decomposing it into a collection of production rule sets each of which has a known VLSI implementation. Those primitive production rule sets correspond to logic gates or standard VLSI cells that are our ultimate building blocks.

The set of operators with which we want to build our circuits is not unique. The descriptions of the operators used in this paper in terms of their production rules and their logic symbols are as follows.

The "and":

$$\begin{align*}
(x, y) \land z & \equiv x \land y \rightarrow z \uparrow \\
& -z \lor \neg y \rightarrow z \downarrow
\end{align*}$$

The "or":

$$\begin{align*}
(x, y) \lor z & \equiv x \lor y \rightarrow z \uparrow \\
& -z \land \neg y \rightarrow z \downarrow
\end{align*}$$

The wire:

$$x \leftrightarrow y \equiv x \rightarrow y \uparrow \\
& -x \rightarrow y \downarrow$$

The inor:

$$x \leftrightarrow (y, z) \equiv x \rightarrow y \uparrow, z \uparrow \\
& -x \rightarrow y \downarrow, z \downarrow$$

The C-element:

$$\begin{align*}
(x, y) \leftrightarrow z & \equiv x \land y \rightarrow z \uparrow \\
& -z \land \neg y \rightarrow z \downarrow
\end{align*}$$

The asymmetric C-element:

$$\begin{align*}
(x, y) \leftrightarrow z & \equiv x \land y \rightarrow z \uparrow \\
& -z \land \neg y \rightarrow z \downarrow
\end{align*}$$

The "flip-flop":

$$\begin{align*}
(x, y) \leftrightarrow z & \equiv x \rightarrow z \uparrow \\
y \leftrightarrow z \downarrow
\end{align*}$$

A negated input or output is represented on the figures by a small circle on the corresponding port. A wire with its input negated is an inverter. A cell with a negated input is considered as one cell, and not as the composition of an inverter and a cell.

5. 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. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program. Process decomposition plays an important role in the compilation of large programs. We won't need it in the example treated here. See [5] for a typical use of this transformation.

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 in terms of a "four-phase handshaking" protocol. Channel $(X, Y)$ is implemented by the two wires $(x \leftrightarrow y)$ and $(y \leftrightarrow z)$.

Initially, $xo$, $xz$, $y$, and $yz$ are false. 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:

$$X : xo \uparrow; [yz]; xo \downarrow; [-yz]$$ (1)

\[ (x, y) \land z \equiv x \land y \rightarrow z \uparrow \]
\[ -z \lor \neg y \rightarrow z \downarrow \]
\[ (x, y) \lor z \equiv x \lor y \rightarrow z \uparrow \]
\[ -z \land \neg y \rightarrow z \downarrow \]
\[ x \leftrightarrow y \equiv x \rightarrow y \uparrow \]
\[ -x \rightarrow y \downarrow \]
\[ x \leftrightarrow (y, z) \equiv x \rightarrow y \uparrow, z \uparrow \]
\[ -x \rightarrow y \downarrow, z \downarrow \]
\[ (x, y) \leftrightarrow z \equiv x \land y \rightarrow z \uparrow \]
\[ -z \land \neg y \rightarrow z \downarrow \]
\[ (x, y) \leftrightarrow z \equiv x \land y \rightarrow z \uparrow \]
\[ -z \land \neg y \rightarrow z \downarrow \]
\[ (x, y) \leftrightarrow z \equiv x \rightarrow z \uparrow \]
\[ y \leftrightarrow z \downarrow \]
\[ Y \equiv \{ y_i \}; \quad \{ y_o \}; \quad \{ \neg y_i \}; \quad \{ y_o \} \quad (2) \]

When no action of a matching pair is probed, the choice of which one should be active out of all passive actions 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 are passive. If \(X\) is used, all \(X\)-actions are active—with the obvious restriction that \(Y\) cannot be used in the same program. The implementation of the probe is simply:

\[ X \equiv x_i \]
\[ Y \equiv y_i \quad (3) \]

A probed communication action \(X \rightarrow \ldots X\) is implemented:

\[ x_i \rightarrow \ldots x_0 \}; \quad \{ -x_i \}; \quad x_0 \downarrow \]

Reshuffling
Consider the handshaking expansion of 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 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. The possibility to reshuffle the second half of the handshaking sequence plays an important role in the compilation method as a source of algebraic manipulations.

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. This is the most difficult step in particular because it requires, in all but trivial cases, the introduction of state variables to identify each state of the computation uniquely.

Operator reduction
The last step, called "operator reduction", consists in identifying sets of production rules in the program with sets of production rules describing operators. The non-trivial part in this step is called "symmetrization". It is used for transforming the guards of the production rules so as to make them 'look like' the guards of operators. After this last step, the program has been replaced by a network of operators for which standard cells exist. (We have constructed a cell library of self-timed elements in SCMOS technology. Since many cells are parameterized, the library is expandable.)

6. Example: Single variable register
Consider the following process that provides read and write access to a simple boolean variable \(x\):

\[ e[V \rightarrow P x \mid \overline{Q} \rightarrow Q x] \quad (4) \]

where \(\neg P \lor \neg Q\) holds at any time, i.e., read and write requests exclude each other in time.

Handshaking expansion
The handshaking expansion of (4) uses the "double-rail" technique: the Boolean value of \(x\) is encoded on two wires, one

\[ \begin{array}{c}
\text{pi2} \\
\text{po} \\
\text{pl1} \\
\end{array} \rightarrow \begin{array}{c}
\text{qo2} \\
\text{qi} \\
\text{qo1} \\
\end{array} \]

\[ \text{Figure 1: Single-bit register} \]

for the value true and one for the value false. Each guarded command of (1) is expanded to two guarded commands:

\[ \begin{array}{c}
\{ [p_1 \rightarrow x \}; \quad [z]; \quad [p_0 \}; \quad \{ \neg p_1 \}; \quad p_0 \downarrow \\
\{ p_2 \rightarrow x \}; \quad [\neg z]; \quad [p_2 \}; \quad p_0 \downarrow \\
\{ x \land q_1 \rightarrow q_0 \}; \quad \{ \neg q_1 \}; \quad q_0 \downarrow \\
\end{array} \]

\[ \downarrow \]

\[ \{ \neg x \land q_1 \rightarrow q_0 \}; \quad [\neg q_1 \}; \quad q_0 \downarrow \quad (5) \]

Production rule expansion
The production-rule expansion of the first two guarded commands gives:

\[ \begin{array}{c}
p_1 \rightarrow x \uparrow \\
p_1 \land x \rightarrow p_0 \uparrow \\
\neg p_1 \rightarrow p_0 \downarrow \\
p_2 \rightarrow x \downarrow \\
p_1 \land p_2 \rightarrow p_0 \uparrow \\
\neg p_1 \rightarrow p_0 \downarrow \end{array} \]

The first and fourth p.r.'s correspond to the flip-flop: \((p_1; p_2) \cap x\). The other p.r.'s can be transformed into:

\[ \begin{array}{c}
(p_1 \land x) \lor (p_2 \land \neg x) \rightarrow p_0 \uparrow \\
\neg p_1 \lor \neg x \rightarrow (\neg p_2 \lor x) \rightarrow p_0 \downarrow \\
\end{array} \]

which is the definition of the IP-cell \((p_1; p_2; x) \mid p_0\). This set of p.r.'s can also be implemented as:

\[ \begin{array}{c}
(p_1, x) \triangle p_0, \\
(p_2, \neg x) \triangle p_0, \\
(p_0, p_0) \land p_0. \\
\end{array} \]

The production-rule expansion of the last two guarded commands of (5) gives:

\[ \begin{array}{c}
x \land q_1 \rightarrow q_0 \uparrow \\
\neg x \lor \neg q_1 \rightarrow q_0 \downarrow \\
\neg x \land q_1 \rightarrow q_0 \downarrow \\
x \lor \neg q_1 \rightarrow q_0 \downarrow, \\
\end{array} \]

which corresponds to the two operators \((x, q_1) \triangle q_0\) and \((\neg x, q_1) \triangle q_2\). The circuit is represented in Figure 1.

7. The lazy stack
A lazy stack is one in which the full elements, i.e., the elements of the stack that contain a piece of data, are not necessarily contiguous. For instance, after a "pop" operation removes a data portion from the top element of the stack, the hole created in the top element is not filled even if some other element of the stack contains data portion. Obviously, we must record
whether a stack element is full or empty. In the implementation
given in [3], a Boolean variable is used for this purpose. Here we
shall use a different coding: a stack element is described as two
programs—one for the empty case, one for the full case—which
call each other in a mutually recursive way.

We restrict ourselves to Boolean data portions. A data
portion is added to a stack element by a command on the input
channel "in". A data portion is removed from a stack element
by a command on the output channel "out". We assume that the
environment never attempts to add portions to a full stack
nor to remove portions from an empty stack. Hence a request
to remove a portion from an empty stack causes the element to
obtain the next data portion from the "rest of the stack". Such
an action uses the input channel "get". Similarly, a request to
add a portion to a full element causes the element to push the
portion it contains to the "rest of the stack". Such an action
uses the output channel "put".

The program for the empty stack element is called $E$. The
program for the full stack element is called $F$. We have

$$
E \equiv \begin{cases}
                   \begin{array}{l}
                     \text{in} \to \text{in}; \text{x}, F \\
                     \text{out} \to \text{get}; \text{out}, E \\
                   \end{array} \\
                   \end{cases}
F \equiv \begin{cases}
                   \begin{array}{l}
                     \text{in} \to \text{put}; \text{in}; \text{x}, F \\
                     \text{out} \to \text{out}; E \\
                   \end{array} \\
                   \end{cases}
$$

(0)

The initialization of an empty stack element is a call of $E$.
The initialization of a full stack element is a call of $F$.

8. Implementation of the control part
Let us first implement the "control part" of the program, i.e.,
the programs $E$ and $F$ from which message communication
has been removed. We assume that the stack is initially
empty. Instead of using mutual recursion, we use (what may look like)
a slightly less symmetrical coding of (0): we introduce the channel
$(t, t')$ and call $F$ from within $E$ by the usual construction of
process decomposition. We get

$$
E \equiv \begin{cases}
                   \begin{array}{l}
                     \text{in} \to \text{in}; t \\
                     \text{out} \to \text{get}; \text{out} \\
                   \end{array} \\
                   \end{cases}
F \equiv \begin{cases}
                   \begin{array}{l}
                     \text{in} \to \text{put}; \text{in} \\
                     \text{out} \to \text{out}; t' \\
                   \end{array} \\
                   \end{cases}
$$

(7)

In the handshaking expansion, the choice of active and
passive communications is entirely dictated by the occurrence
of the probes. We get

$$
E \equiv \\
F \equiv 
$$

(8)

9. Compilation of $E$
The first guarded command of $E$ is a standard passive-active
buffer element implemented as an active-active buffer composed
with a passive-passive adaptor (Fig. 2a). The second guarded
command is a standard stack element implemented as an active-
active buffer with input $outi$ inverted (Fig. 2b). The active-
active buffer is a standard cell called a $D$-element.

Next, we have to enforce mutual exclusion between the
two guarded commands of $E$. Since in and out are mutually
exclusive, it suffices to guarantee that when an in is completed
in the first guarded command, the second guarded command
cannot start until $t$ is completed. In order to strengthen the
guard of the second command with the appropriate expression,
we introduce in the handshaking expansion of the first guarded
command the variable $x$. We get

$$
x \land \text{ini} \to \text{ino} \uparrow; \text{in}; \text{in}; \text{out}; \text{out}; \text{to} \uparrow; \text{to}; \text{to} \uparrow; \text{to} \uparrow
$$
as the handshaking expansion of the first guarded command.
Obviously, it suffices to strengthen the guard of the second
guarded command with $x$ to guarantee mutual exclusion be-
tween the two g.c.'s. We get

$$
\text{out}; \text{out}; \text{to} \uparrow; \text{to} \uparrow; \text{to} \uparrow; \text{to} \uparrow
$$

Since we can weaken $\neg \text{out}$ as $\neg \text{out} \lor \neg \neg \text{out}$, the only transformation is the replacement of $\text{out}$ by $x \land \text{out}$. This gives the
circuit of Figure 3 as an implementation of $E$.

10. Compilation of $F$
The compilation of the first guarded command of $F$ is identi-
cal to that of the second command of $E$, with the appropriate
change of variables. The compilation of the second command,
however, can be drastically simplified by reshuffling. Since
channel $(t, t')$ is an internal channel, we can reshuffle the
handshaking sequence of $t'$ without deadlock. The handshaking
expansion of the second guarded command becomes:

$$
t' \land \text{out}; \text{out}; \text{to} \uparrow; \text{to} \uparrow; \text{to} \uparrow; \text{to} \uparrow
$$

This sequence compiles immediately into the C-element:
$(t', \text{out}) \subseteq (\text{out}, t')$. 
The channels in and out are used both in E and F, so we need to merge the local copies of in and the local copies of out in the standard way. The resulting circuit for the control part of the stack element is shown in Figure 4.

11. Implementation of the data path
Let S1 and S2 denote program (6) and program (7), respectively. We now have to extend the implementation of S2 so as to obtain an implementation of S1. We want to leave S2 unchanged and introduce an extra “data path” process P such that the parallel composition of S2 and P implements S1. More precisely, the channels in, out, get, put of S2 are renamed in', out', get', put'. P communicates with S2 via the renamed channels and with the environment via in, out, get, put. (See Figure 5.)

By comparing S1 and S2, we derive that P has to implement the operations:

\[
\begin{align*}
\text{in}' \cdot \text{in}\1 \cdot \text{in}\2 \\
\text{out}' \cdot \text{out}\1 \cdot \text{out}\2 \\
\text{get}' \cdot \text{get}\1 \\
\text{put}' \cdot \text{put}\1 \\
\end{align*}
\]

where \(A \cdot B\) denotes the simultaneous execution of \(A\) and \(B\).

(We can define the completion of an action so that the simultaneous execution of two actions is well-defined. The implementation of \(A \cdot B\) amounts to interleaving the handshaking sequences of \(A\) and \(B\).)

The implementation of the four actions of \(P\) is based on the register program constructed in Section 6. For the sake of brevity, we omit the rest of the derivation which can be found in [8]. The entire data path is described in Figure 6.

The dual-port flip-flop used in the data path is defined as:

\[
\{a_1, a_2; t_1, t_2\} \| z \quad \iff \quad (a_1 \lor a_2) \implies \begin{cases} 
\begin{align*}
&z \lor t_1 \\
&\neg t_2
\end{cases}
\end{align*}
\]

(By definition, at most one input is true at any time.)

12. The complete circuit
Two important optimizations are added to the design. The first one concerns the implementation of the second guard of \(E\):

\[
\text{out} \rightarrow \text{get}\2; \text{out}\2.
\]

We observe that, in this case, unlike all other guarded commands of (6), the value of \(x\) involved in the second action (out\2) is the same as the value of \(x\) involved in the first action (get\2). We can therefore encode the value of \(x\) in the handshaking expansion of the guarded command without having to use the register. The restfulen handshaking expansion including the double-rail encoding of \(x\) gives:

\[
\neg \text{in} \land \text{out} \rightarrow \text{get}\1; \quad [\text{get}\1 \rightarrow \text{out}\1] \quad [\text{get}\2 \rightarrow \text{out}\2];
\]

\[
[\neg \text{out}]; \quad \text{get}\1; \quad [\neg \text{get}\1 \rightarrow \text{out}\1] \quad [\neg \text{get}\2 \rightarrow \text{out}\2].
\]

The circuit is:

\[
(\neg \text{in}; \neg \text{out}) \land \text{get}\1
\]

\[
\text{get}\1 \equiv \text{out}\1
\]

\[
\text{get}\2 \equiv \text{out}\2
\]

The second optimization concerns the implementation of \(\text{in}' \cdot \text{in}\2\), which is more complex than that of \(\text{get}' \cdot \text{get}\2\) because in\2 is passive while get\2 is active. We replace in\2 and put\2 by in\2; in\2 and out\2; out\2, respectively, with in\2 passive and in\2 active, and out\2 active and out\2 passive. For the output action out, the implementation is the same whether the channel is active or passive. The complete circuit is shown in Figure 7 with the data path extended to four bits.

13. Concluding remarks
By combining control and data, the design of a lazy stack encompasses most self-timed design issues (except for arbitration which is treated in [4] and [5]).

Let us summarize the main advantages of the method. First, the source language, in particular the use of the probe,
produces compact and efficient algorithms, which can be further “tuned” through process decomposition. Second, the handshaking expansion combined with reshuffling offers powerful algebraic manipulations. Third, the production rule notation provides a canonical representation of the circuit which is straightforward to translate in whatever set of VLSI gates is available or convenient to use. Finally, the notion of stability of a guard captures exactly the necessary and sufficient condition to avoid races and hazards.

We already have a compiler that produces about the same design fully automatically [1]. Figure 8 shows a typical layout produced by the assembler from the operator set. Each operator has a standard cell representation. The cells of a process are stacked to form a tower in which power, reset, and ground run vertically.

ACKNOWLEDGEMENTS are due to Steve Burns for his contribution to the design of the stack, and to Cal Jackson for his help in the preparation of the manuscript.

REFERENCES

Figure 7: Stack element with four-bit data path

Figure 8: Layout of the control part