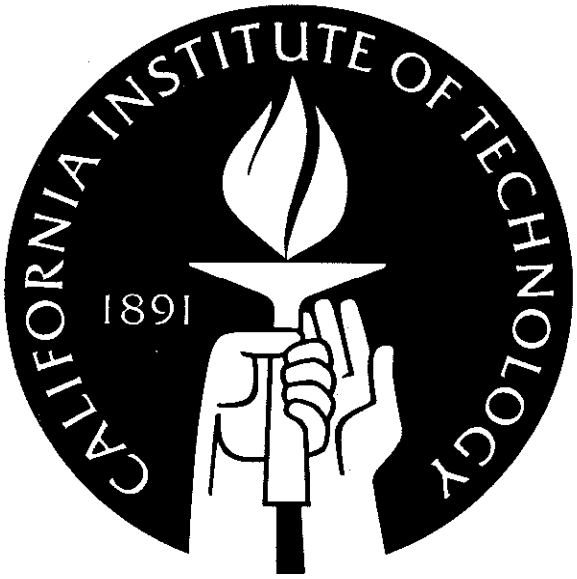

The Limitations to Delay-Insensitivity in Asynchronous Circuits

Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-90-02

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JAN 1990		**00-01-1990 to 00-01-1990**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Limitations to Delay-Insensitivity in Asynchronous Circuits	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**19**	

THE LIMITATIONS TO DELAY-INSENSITIVITY IN ASYNCHRONOUS CIRCUITS

Alain J. Martin

to appear in:
Proceedings of the 6th MIT Conference on Advanced Research in VLSI
MIT Press, 1990

Nov. 1989, Revised Jan. 1990

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-90-02

The Limitations to Delay-Insensitivity in Asynchronous Circuits

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

Asynchronous techniques—that is, techniques that do not use clocks to implement sequencing—are currently attracting considerable interest for digital VLSI circuit design [3, 4, 6, 8, 1, 11], particularly when the circuits produced are *delay-insensitive* (DI). A digital circuit is DI when its correct operation is independent of the delays in operators and in the wires connecting the operators, except that the delays are finite and positive.

In this paper, we characterize the class of circuits that are entirely DI, and we show that this class is surprisingly limited: Practically all circuits of interest fall outside the class since closed circuits inside the class may contain only C-elements as multiple-input operators.

The paper is organized as follows: First, we introduce the *stable gate* model of DI circuits, which is based on the notion of *production rules* (PRs) as elementary computation steps. We then define a partial ordering on transitions in the circuits. We prove that all DI circuits have to fulfill the so-called *Unique-Successor-Set* criterion; and we show that the class of circuits that meet this criterion is very limited. We also give a characterization of the class of computations that admit a DI implementation. Finally, we discuss what we consider to be the weakest compromise to delay-insensitivity, namely, *isochronic forks*.

1 Circuits as Networks of Gates

A DI circuit is a network of logical operators, or *gates*. A gate has one or more Boolean inputs and one Boolean output. (Later, we will introduce gates with multiple outputs.) The state of the circuit is entirely characterized by the values of the input and output variables of the gates.

We assume that all circuits are *closed*: Each variable of a circuit is the input of a gate and also the output of a gate. An open circuit is transformed into a closed one by representing the environment of the circuit as gates.

1.1 Gates as Pairs of Production Rules

The simple assignments $x := true$ and $x := false$ are denoted by $x\uparrow$ and $x\downarrow$, respectively. An execution of a simple assignment is called a *transition*. The *result* of a transition of type $x\uparrow$ is the postcondition x (standing for the predicate "x is true"); the result of a transition of type $x\downarrow$ is the postcondition $\neg x$ (standing for the predicate "x is false").

A gate with output variable z is defined by the two *production rules*:

$$\begin{aligned} B_u &\mapsto z\uparrow \\ B_d &\mapsto z\downarrow \end{aligned}$$

where B_u is the condition on the input variables for a transition of type $z\uparrow$ to take place, and B_d is the condition on the input variables for a transition of type $z\downarrow$ to take place. B_u and B_d are called the *guards* of the PRs. They are Boolean expressions in terms of the Boolean variables of the circuit.

We will assume that a guard is in disjunctive-normal form, that is, it is either a *literal*, a *term*, or a disjunction of terms. A literal is a variable or its negation; a term is a conjunction of literals.

EXAMPLES: The gate, with inputs x and y, and output z, defined by the two PRs

$$\begin{aligned} x \wedge y &\mapsto z\downarrow \\ \neg x \vee \neg y &\mapsto z\uparrow \end{aligned}$$

is usually called a "nand-gate." The gate, with inputs x and y, and output z, defined by the two PRs

$$\begin{aligned} x \wedge y &\mapsto z\uparrow \\ \neg x \wedge \neg y &\mapsto z\downarrow \end{aligned}$$

is usually called a "Muller-C element." □

1.2 Non-interference and Stability of PRs

Obviously, the simultaneous execution of both PRs of a gate would result in a malfunctioning of the circuit. Hence, the two PRs of a gate must fulfill the *non-interference* requirement.

Non-interference. $\neg B_u \vee \neg B_d$ *is invariantly true.*

A gate is a partial function when the non-interference requirement is not a tautology but has to be maintained as a program invariant. The flip-flop is an example of such a gate.

The non-interference requirement eliminates the most obvious case of malfunctioning of a gate. But other forms of malfunctioning, usually

called *hazards*, have to be eliminated as well. A hazard is an incomplete transition on the output of a gate caused either by two consecutive transitions on one input variable or by some concurrent changes on several input variables. In our model, all occurrences of hazards are eliminated as follows.

At any time, all PRs of a circuit with a true guard are executed concurrently. The net effect of the execution of a PR is to establish the result of the PR; therefore, the execution of a PR is considered to be correctly terminated when the result holds. (The *result of a PR* is the result of the transition caused by an execution of the PR.) The execution of a PR correctly terminates unless the guard is falsified before the result holds. In that case, the net effect of the execution is undefined. We therefore add a requirement called *stability*.

Stability. *The guard of a PR is stable in a computation if it is falsified only in states where the result of the PR holds.*

(The stability of the physical implementation of a PR also requires that the changes in value of the physical quantity—voltage, in MOS technology—representing the Boolean values be *monotonic*. However, monotonicity around the stable values is, in general, neither attainable, because of noise, nor necessary.)

If a circuit fulfills the non-interference and stability criteria, no glitch or hazard can corrupt the value of the variables. At any point in time, the physical quantity representing a variable either has one of the two stable values representing the two Boolean values, or is monotonically changing from one stable value to the other.

Observe that, whereas non-interference can be a property of either the gate—if $\neg B_u \vee \neg B_d$ is a tautology—or of the circuit using the gate, the stability of a gate is always a property of the circuit.

Any pair of PRs that set and reset the same output variable defines a valid gate, with the exception of *self-invalidating* PRs. A rule with guard g and result r is self-invalidating if $r \Rightarrow \neg g$ may hold as a post-condition of a transition of that rule. In other words, the execution of the rule may falsify the guard. For example, the rules $x \mapsto x\downarrow$ and $\neg x \mapsto x\uparrow$ are self-invalidating.

The execution of a PR in a state where the result holds is called *vacuous*; otherwise, it is called *effective*. From the definition of the execution of a PR, the vacuous execution of a PR is equivalent to a *skip*. Consequently, it is always possible to modify the guard of a PR so that it does not contain the output variable of the gate. (This is achieved by removing all terms that contain the result as literal. For example, $(x \wedge z) \vee y \mapsto z\uparrow$ can be replaced with $y \mapsto z\uparrow$, since an execution of the PR in the state where $x \wedge z$ holds is vacuous.)

Hence, gates do not contain variables that are both input and output (self-loops). In the sequel, unless specified otherwise, an execution of a PR is an effective execution.

1.3 Wires

A priori, a wire with input x and output y is the gate defined by the PRs $x \mapsto y\uparrow$ and $\neg x \mapsto y\downarrow$. But, since the composition of any gate, *including a wire*, with a wire is the gate itself with one of its variables renamed, we can add an arbitrary number of wire gates to a circuit definition without actually changing the circuit. In order to have a unique network of gates for each circuit, we exclude the wire from the gates; a wire is just a renaming mechanism for variables.

So far all gates except the wire have more inputs than outputs, but most circuits have as many outputs as inputs. We must therefore reset the balance by introducing at least one gate with more outputs than inputs. This gate is the *fork*.

1.4 Forks and Multiple-Output Gates

A fork has one input and at least two outputs. The fork, f, with input x and outputs y and z is defined as

$$\begin{array}{ll} x & \mapsto y\uparrow, z\uparrow \\ \neg x & \mapsto y\downarrow, z\downarrow \end{array}$$

where the comma means the execution of the two assignments in any order or concurrently. The generalization to an arbitrary number of outputs is obvious. The gate

$$\begin{array}{ll} B_u & \mapsto x\uparrow \\ B_d & \mapsto x\downarrow \end{array}$$

composed with fork f is equivalent to the gate with outputs y and z

$$\begin{array}{ll} B_u & \mapsto y\uparrow, z\uparrow \\ B_d & \mapsto y\downarrow, z\downarrow . \end{array}$$

Hence, the fork is just a mechanism for replicating the outputs of a gate and for defining gates with an arbitrary number of outputs. The following discussion is somewhat simplified if we eliminate the fork and allow instead the type of multiple-output gates that correspond to the composition of a single-output gate and a fork. But gates defined in this way have an important restriction: *The effective execution of a PR of a gate contains an effective transition on each output of the gate.*

1.5 Summary of the Model

The only restriction that these definitions and conventions introduce on the class of circuits being considered is the exclusion of gates with self-loops and of arbitration devices. Unlike models based on the "fundamental mode" of operation, several inputs of a gate may change values simultaneously as long as the stability of the guards of the PRs is preserved.

Also, we do not assume that the transitions are instantaneous: A variable value changes monotonically from the "bottom" value representing one logical value to the "top" value representing the other logical value, and vice-versa. Because the transitions durations are finite but positive and variable, the ordering of transitions in a circuit has to be defined with care.

2 Partial Order of Transitions

The specification of a sequential circuit defines a partial order of actions taken from a repertoire of commands. In order to assert that a circuit fulfills a specification, we must relate this partial order to some other order relation among transitions of the circuit. The partial order of transitions is defined as follows.

Consider an effective execution of a PR causing the transition t, and let C be a term of the guard such that C holds for this execution of the PR.

We attach to C a set, T, of transitions in the following way. Each literal of C uniquely defines a transition: The literal x is the result of a transition of type $x\uparrow$, and the literal $\neg x$ is the result of a transition of type $x\downarrow$. (The initialization of a variable is also considered a transition.) *By definition, we say that transition t is a successor of each transition of T.* In other words, a transition is the successor of the set of transitions that make the guard true, including initializations.

For example, if the PR is $x \wedge y \mapsto z\uparrow$, we say that each transition $z\uparrow$ is the successor of a transition $x\uparrow$ and of a transition $y\uparrow$.

If the guard of the PR is of the form $A \vee B$, the transition is the successor of the set of transitions that make A true, or of the set of transitions that make B true. Hence, the successor relation defined is not unique for a given circuit. A *computation* is a particular successor relation on a set of transitions, such that each computation corresponds to a possible execution of the circuit. The set of transitions of a computation is finite if the corresponding execution of the circuit terminates, and possibly infinite otherwise.

From the successor relation, we can now construct a relation $\prec$ that is a pre-order; that is, it is transitive and anti-reflexive. Once we have

the pre-order relation $\prec$, we construct the partial order $\preceq$ by defining $t1 \preceq t2$ to mean $t1 \prec t2$ or $t1 = t2$.

Transitivity. *For any two transitions $t1$ and $t2$, we say that $t1 \prec t2$ when $t2$ is a successor of $t1$, or there exists a transition $t3$ such that $t1 \prec t3$ and $t3 \prec t2$.*

Anti-reflexivity. *$t \prec t$ holds for no transition t.*

REMARK: Anti-reflexivity is satisfied if, for each ring of gates in the circuit, there is always at least one PR whose guard is true and whose result is false—the ring "oscillates." Anti-reflexivity excludes rings of gates that are used to maintain constant values of variables, as in cross-coupled device constructions of storage elements. We therefore assume that the storage elements are parts of "perfect wires," so to speak, that keep the value of a variable until the next transition on the variable. □

Definition. *A chain from a to b is a finite, non-empty set $\{t_i, 0 \leq i < n\}$ of transitions such that $t_0 = a$, $t_n = b$, and for all i, $0 < i < n$, t_i is a successor of t_{i-1}. By construction, $a \preceq b$ means that there is a chain from a to b. If $a \prec b$, we say that b follows a.*

3 Implementation of Stability

Consider again an execution of a PR with guard B and transition t. Either B is never falsified once it holds, but then t is the last transition on the variable involved, and we say that the transition is *final.* Or B is falsified after a finite number of transitions following t, in which case, in order to implement the stability of B, we have to see to it that t is completed before B is falsified.

For all transitions i that falsify B, we have to guarantee $t \prec i$. Hence, by definition of the order relation, there must be a transition s such that s is a *successor* of t, and $s \preceq i$. We say that s *acknowledges* t. Hence, the

Acknowledgment Theorem. *In a DI circuit, each non-final transition has a successor transition.*

By construction of multiple-output gates, we have the

Corollary. *In a DI circuit, a non-final transition on an input of a gate has a successor transition on each output of the gate.*

EXAMPLE: Consider the three following gates with two inputs, x and y, and one output, z. The *flip-flop* is defined as $x \mapsto z\uparrow$ and $\neg y \mapsto z\downarrow$, the *asymmetric C-element* as $x \wedge y \mapsto z\uparrow$ and $\neg y \mapsto z\downarrow$, and the *switch* as $x \wedge y \mapsto z\uparrow$ and $x \wedge \neg y \mapsto z\downarrow$.

Since no guard of these gates has a term containing the literal $\neg x$, a transition of type $x\downarrow$ has no successor. Hence, according to the Acknowledgment Theorem, there can be at most two transitions on x in any computation of a DI circuit using any of these three gates. □

4 The Unique-Successor-Set Criterion

Later on, we shall give a simple criterion for deciding whether a given circuit—a network of gates—is DI. But such a criterion does not tell us whether there exists a DI circuit for a given specification. We shall therefore formulate a more general theorem that characterizes the partial orders of transitions that admit a DI implementation. This criterion enables us to decide that a program has no DI implementation without having to construct a circuit.

Successor Set. *In a computation, the successor set of a transition t is the set of variables x such that a transition on x is a successor of t.*

Unique-Successor-Set Property. *A computation has the unique-successor-set (USS) property when all non-final transitions on the same variable have the same successor set. A set of computations has the USS property when all non-final transitions on the same variable have the same successor set in all computations of the set.*

Unique-Successor-Set Theorem. *A set of computations of a DI circuit has the USS property.*

Proof. Consider an arbitrary variable x of a DI circuit. By the corollary of the Acknowledgment Theorem, any non-final transition t on x has a successor transition on each output of the gate, say G, of which x is an input.

By definition of the successor set, the set of output variables of G is the successor set of t. But since the set of output variables of a gate is unique, the successor set is the same for all non-final transitions on x. □

5 Characterization of DI computations

Although the Unique-Successor-Set Theorem is a direct consequence of the Acknowledgment Theorem, its formulation in terms of computations instead of gates makes it possible to lift the result from the implementation level to the specification level. Since the partial orders of actions defining a circuit are projections of the partial orders of actions implementing it, we shall investigate whether the USS property is maintained by projection.

Definition. *Given a computation, c, on a set of variables, V, the projection of c on a subset, W, of V is the computation derived from c by removing all transitions on variables of $V \backslash W$ from the chains of c. The projection of a set of computations is the set obtained by projecting each element of the original set.*

Projection Theorem. *If a set of computations has the USS property, then its projection on a subset of variables has the USS property.*

Proof. By definition, the projection of a set of computations on W can be obtained by removing the elements of $V \backslash W$ one for one from all chains of each computation of the set. We prove the theorem by showing that removing all transitions on one variable, say, w, maintains the USS property of the set.

Let x be another variable, and let X be the USS of (all transitions on) x in all computations of the set. Either w does not belong to X and X is left unchanged by the transformation, or w is removed from X. But then, for each transition tx on x, the successor set of the transition on w that follows tx must be added to the successor set of tx. Since all transitions on w have the same successor set in all computations of the set, the new X is the same for all transitions and all computations of the set. □

5.1 Example: One-Place Buffer

The cyclic program $*[X; Y]$, where X and Y are communication commands, is called a *one-place buffer*[1]. It is a basic building block of asynchronous circuit design since it is used to implement the sequencing of any two actions. With a four-phase handshaking protocol for implementing the communications, an expansion of the program in terms of elementary variables is:

$$*[[xi]; xo\uparrow; [\neg xi]; xo\downarrow; yo\uparrow; [yi]; yo\downarrow; [\neg yi]],$$

where xi and yi are the input variables, and xo and yo are the output variables[2]. (See Figure 1.) The environment of the circuit can be simply modeled as the two programs:

$$*[xi\uparrow; [xo]; xi\downarrow; [\neg xo]]$$
$$*[[yo]; yi\uparrow; [\neg yo]; yi\downarrow].$$

These three programs are concurrent. Now observe that the projection of a computation on the output variables of the first program gives the

[1]The notation $*[S]$ stands for the non-terminating repetition of the program S.

[2]For an arbitrary Boolean expression B, the command $[B]$ is a shorthand notation for $[B \rightarrow skip]$, and can be informally defined as "wait until B holds."

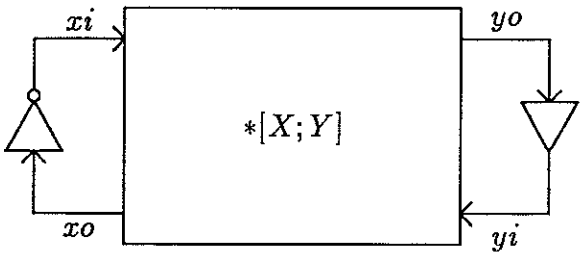

Figure 1: A one-place buffer and its interface

computation described by the program

$$*[xo\uparrow; xo\downarrow; yo\uparrow; yo\downarrow].$$

Obviously, this computation does not have the USS property; therefore, by the Projection Theorem, the closed circuit implementing the three programs is not DI. But the two environment programs can be implemented with an inverter gate and an identity gate, which are DI circuits. Hence, there is no DI circuit implementing this version of the one-place buffer with four-phase handshaking.

We can state a more general result. We observe that, for whatever four-phase handshaking is chosen for X and Y, the projection on the output variables is always $*[xo\uparrow; xo\downarrow; yo\uparrow; yo\downarrow]$, unless the handshaking actions of X are reordered ("shuffled") with respect to the handshaking actions of Y. Hence, the

Theorem. *There is no DI circuit implementing a one-place buffer with unshuffled four-phase handshaking.*

We can shuffle the handshaking actions of X with respect to the handshaking actions of Y, so that the projection on the output variables is the sequence

$$*[xo\uparrow; yo\uparrow; xo\downarrow; yo\downarrow].$$

Now, the sequence has the USS property, and we can implement the one-place buffer as a DI circuit. An example is shown in Figure 2.

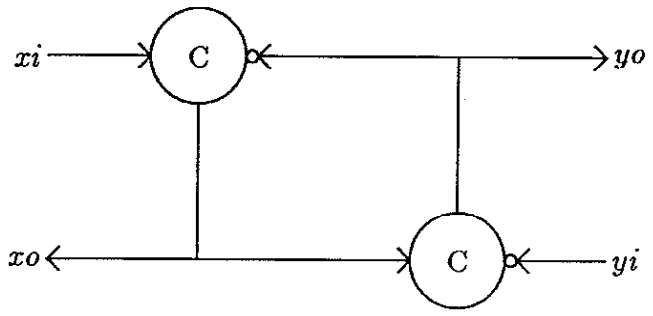

Figure 2: A DI circuit for the one-place buffer

6 Specifications and the USS Property

The Projection Theorem is very useful because we can also define when a specification has the USS property. If a specification does not have the property, we can immediately conclude that there exists no DI implementation of the specification. The projection from implementation to specification occurs as follows.

We assume that, whatever specification notation is used, whether programs, traces, or regular expressions, it is possible to derive from the specification certain properties of the partial order of actions involved. Hence, in the sequel, a *specification* is a set of partial orders of actions, where an action is an execution of a command taken from some given repertoire.

We also assume that an elementary variable can be uniquely identified with (the implementation of) each command: The transitions on the variable occur only in the executions of the command, and each execution of the command contains a transition on the variable. This (in theory, slightly restrictive) assumption is needed only for the following

Specification Theorem. *If the specification of a circuit does not have the USS property, the circuit is not DI.*

Proof. Consider a specification, S, of a circuit. For each command, X, of S, we substitute a transition on the elementary variable x that is uniquely associated with X. We obtain a set, s, of partial orders

of transitions on elementary variables. Since the existence of the USS property is independent of whether the transitions are upgoing or downgoing (that is, the "direction" of the transitions), we can decide whether s has the USS property even though the direction of the transitions in s is undefined.

By definition, we say that specification S has the USS property if and only if the set, s, thus defined has the USS property. By construction, s is a projection of the set of computations of the circuit specified by S. Hence, by the Projection Theorem and the USS Theorem, if s does not have the USS property, the circuit is not DI. □

EXAMPLES: The following examples, which we give without proofs, show how limited is the class of programs that admit a DI implementation. (In the examples, all commands are different from *skip*.) We assume that the semantics of the program notation are clear enough that we can identify the programs with the partial order of actions they represent.

• Let $P \equiv *[S_1; S_2; \ldots S_n]$, and assume that there is no equivalent program

$$*[S_1; S_2; \ldots S_k]$$

with $k < n$. (We say that P is a minimal representation. For instance, $*[X; X]$ is not minimal since $*[X]$ is an equivalent program.)

Then P has the USS property if and only if $S_i \neq S_j$ for $i \neq j$. Hence, the "modulo-2 counter" $*[X; X; Y]$ and all other "modulo-k counters" have no DI implementation. A similar result has been proved by C. J. Seger[9].

• The program $*[S_1; [B_1 \rightarrow S_2 [] B_2 \rightarrow S_3]; S_4]$, with $S_2 \neq S_3$, does not have the USS property. Hence, there is no DI circuit implementing such a selection command. □

7 Gate Characterization of DI Circuits

We have already seen that, apart from the trivial case where one input of the gates changes at most twice, there is no DI circuit that contains either a flip-flop, or an asymmetric C-element, or a switch. In the same way, we can use the USS and the Projection Theorems to show that there is no DI circuit containing either an *or-gate*, or an *and-gate*, or an *exclusive-or*, in which each input of the gates changes more than a minimum number of times specific to each case. Consider an or-gate with inputs x and y and output z. The only sequence[3] in which each

[3]The notation $(S)^*$ is the Kleene-star notation standing for an arbitrary number of actions S in sequence.

transition on an input is acknowledged is:

$$((x\uparrow; z\uparrow; x\downarrow; z\downarrow)^*; (y\uparrow; z\uparrow; y\downarrow; z\downarrow)^*)^*$$

We easily see that any computation that contains a transition on both inputs does not have the USS property.

The cases of the and-gate and of the exclusive-or are treated similarly and are left as an exercise for the reader. After having eliminated all gates with at most two inputs except the inverter and the Muller-C element, we are led to conjecture that a DI circuit contains only C-elements. C-elements are defined as follows.

Definition. *An n-input gate in which B_u is the conjunction of the n input variables and B_d is the conjunction of the negations of the n input variables is called an n-input C-element. A gate derived from a C-element by negating one or more literals in B_u or B_d is also a C-element.*

The Muller-C element is a two-input C-element according to our definition. A one-input C-element reduces to either a wire or an inverter.

C-Element Theorem. *If a DI circuit has only one computation, and if the computation contains at least three transitions on each variable, then the circuit can be constructed with C-elements only.*

Proof. Let x be an arbitrary variable of the circuit; x is the input of gate g with output z. We shall prove that g can be implemented as a C-element. Since there are no self-loops, x and z are different variables.

First, observe that because of the non-interference, all transitions on the same variable are totally ordered. And because all transitions are effective, upgoing and downgoing transitions on the same variable alternate.

Since the circuit contains at least three (effective) transitions on each variable, at least one transition of type $x\uparrow$ is followed by a transition of type $x\downarrow$, and at least one transition of type $x\downarrow$ is followed by a transition of type $x\uparrow$.

Let $t1$ be a transition of type $x\uparrow$ and $t2$ be the transition of type $x\downarrow$ following it. For the guard of the PR of $t1$ to be stable, there must be a transition tz on z such that $t1 \prec tz \prec t2$. We also know that tz is a successor of $t1$.

By the USS Theorem and the Projection Theorem, there is exactly one transition tz on z such that $t1 \prec tz \prec t2$. By the same argument, there is exactly one transition on z between a transition of type $x\downarrow$ and the transition of type $x\uparrow$ following it.

Without loss of generality, assume that the first transition on x is of type $x\uparrow$ and the first transition on z is of type $z\uparrow$. Then, because of the

alternation of upgoing and downgoing transitions on each variable, each transition of type $z\uparrow$ is the successor of a transition of type $x\uparrow$, and each transition of type $z\downarrow$ is the successor of a transition of type $x\downarrow$.

By definition of the successor relation, x holds as a precondition of each transition $z\uparrow$; thus, guard B_u of g can be formulated so that all terms contain x, since a term that is never true can be removed. Hence, B_u can be chosen of the form $x \wedge C_u$, where C_u does not contain x. Symmetrically, guard B_d of g can be chosen of the form $\neg x \wedge C_d$, where C_d does not contain x. Since this property of B_u and B_d holds for each input of g, g is a C-element or can be replaced with a C-element. □

8 Isochronic Forks

Since the class of DI circuits is so limited, we must have compromised the delay-insensitivity in the circuits that we designed using the synthesis method described, for instance, in [5] and [4]. Let us analyze a standard sequencing circuit used in this design style. (It is similar to the one-place buffer, but is simpler to use as an example.) This circuit (Figure 3) is an implementation of the sequence of elementary actions:

$$*[[xi]; yo\uparrow; [yi]; u\uparrow; [u]; yo\downarrow; [\neg yi]; xo\uparrow; [\neg xi]; u\downarrow; [\neg u]; xo\downarrow].$$

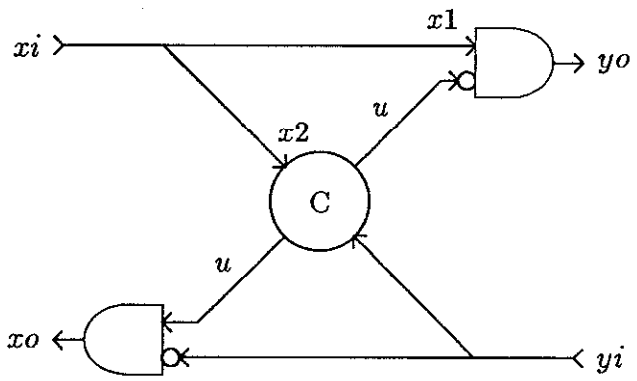

Figure 3: A sequencing element containing isochronic forks

The environment of the circuit is the same as that of the one-place buffer. The x- and y-variables are each parts of a four-phase handshaking sequence, and u is a state variable—without u, it would not

be possible to encode each state of the circuit uniquely. Since the projection of this sequence on the variables xo, yo, and u lacks the USS property, and since the environment of the circuit can be implemented as an inverter and an identity, the circuit is not DI.

In order to find out where we have cheated, we must look at the forks. We observe that xi is an input both of the and-gate with output yo and of the C-element. Hence, the circuit actually contains a fork with input xi and two outputs, say, $x1$ and $x2$. Similarly, the circuit contains a fork with input yi, and a fork with input u. Let us analyze the behavior of the first fork by introducing it explicitly into the set of PRs of the circuit. For the sake of simplicity, we ignore the other two forks. We get:

$$\begin{array}{ll} xi & \mapsto x1\uparrow, x2\uparrow \\ x1 \wedge \neg u & \mapsto yo\uparrow \\ x2 \wedge yi & \mapsto u\uparrow \\ \neg x1 \vee u & \mapsto yo\downarrow \\ \neg yi \wedge u & \mapsto xo\uparrow \\ \neg xi & \mapsto x1\downarrow, x2\downarrow \\ \neg x2 \wedge \neg yi & \mapsto u\downarrow \\ yi \vee \neg u & \mapsto xo\downarrow \end{array}$$

Transitions $x1\uparrow$ and $x2\uparrow$ are both acknowledged by the two PRs that follow. But only transition $x2\downarrow$ is acknowledged. Transition $x1\downarrow$ is *not* acknowledged. Hence, the circuit is not DI, because the Acknowledgment Theorem is not satisfied. Therefore, the completion of transition $x1\downarrow$ is not guaranteed unless we implement the fork as an *isochronic fork*, which is defined as follows.

In an isochronic fork, when a transition on one output is acknowledged, and thus completed, the transitions on all outputs are acknowledged, and thus completed.

(We leave it as an exercise to the reader to check that the fork with input yi must also be isochronic, but not the fork with input u.)

The implementation of an isochronic fork relies on two types of assumptions about delays. First, we have to assume that the difference between the delays in the branches of the fork is negligible compared to the delays in the gates. This requirement is easy to meet in current MOS technology except when there is an inverter on one branch of the fork and not on the other branch(es). The fork with input yi has such an inverter, and therefore, the inverter must be removed by proper circuit transformations.

Second, and more important in current technology, we have to assume that the switching thresholds in the different gates to which the fork is an input are close enough to each other. This requirement is

more difficult to meet than the first one because, on the one hand, the thresholds of individual transistors are difficult to control—in particular in CMOS; on the other hand, the switching thresholds of a gate vary greatly with the logical design of the gate. For these reasons, this requirement may impose a design style in which all gates are implemented as combinational gates, so that the fight between pull-up and pull-down during the switching of the gate keeps the switching threshold around *VDD/2.* Observe that, unlike what is advocated in other compromises to delay-insensitivity, enforcing the locality of the wires offers little help in implementing isochronicity because locality is irrelevant to the issue of threshold voltages!

9 For Whom the Bell Tolls?

Are these results tolling the bell for DI design? Actually, not. At worst, they may slightly embarrass those researchers who claim to have a design method for entirely DI circuits. At best, they vindicate the compromises to delay-insensitivity adopted by several asynchronous design methods. Most likely, they are sobering reminders of the difficulty of VLSI design and the novelty of asynchronous design.

We have proved elsewhere that extending a standard repertoire of DI gates with isochronic forks is sufficient to construct any circuit of interest. The proof consists in giving a circuit implementation for each construct of the progam notation we use (see [2]). I believe the isochronic fork to be the weakest possible compromise to delay-insensitivity in the sense that all other compromises also include isochronic forks: For instance, in speed-independent design[7], all forks are supposed to be isochronic; in self-timed design[10], all forks inside a certain region—called an *equipotential region*—are assumed to be isochronic.

Acknowledgments

The results on the influence of threshold voltages on the functioning of isochronic forks are based on analysis and simulation done by Steve Burns. The formulation of the C-element Theorem in terms of three transitions on each variable is due to a suggestion from Pieter Hazewindus. Acknowledgment is also due to Dražen Borković, Steve Burns, Peter Hofstee, Marcel van der Goot, Tony Lee, and José Tierno for their comments and criticisms. The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745.

References

[1] J.A. Brzozowski and J.C. Ebergen. Recent Developments in the Design of Asynchronous Circuits. Research Report CS-89-18, Computer Science Department, University of Waterloo, 1989.

[2] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. *Proc. Fifth MIT Conference on Advanced Research in VLSI*, ed. J. Allen and F. Leighton, MIT Press, 35-40, 1988.

[3] David L. Dill. Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. MIT Press, 1989.

[4] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, P.J. Hazewindus. The Design of an Asynchronous Microprocessor. *Decennial Caltech Conference on VLSI*, ed. C.L. Seitz, MIT Press, 351-273, 1989.

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

[6] Teresa H. Meng, Robert W. Brodersen, David G. Messerchmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Trans. on CAD*, 8:11, 1185-1205, 1989.

[7] Raymond E. Miller. *Switching Theory*, Vol. 2, Wiley, 1965.

[8] J. Staunstrup and M.R. Greenstreet. Designing Delay-Insensitive Circuits using "Synchronized Transitions." *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[9] Carl-Johan Seger. On the Existence of Speed-Independent Circuits. Research Report CS-87-63, Computer Science Department, University of Waterloo, 1987.

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

[11] Charles L. Seitz. Let's Route Packets Instead of Wires. *These Proceedings.*