UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Recent Developments in the Design of Asynchronous Circuits

J.A. Brzozowski
J.C. Ebergen

Research Report
CS-89-18

May, 1989

Recent Developments in the Design of Asynchronous Circuits*

J.A. Brzozowski and J.C. Ebergen

Computer Science Department
University of Waterloo
Waterloo, Ont., Canada N2L 3G1

Abstract

Some recent developments in the design of asynchronous circuits are surveyed. The design process is considered in two parts. First, the communication behaviour of the component to be designed is formally specified and this specification is decomposed into a network of basic components. Second, the basic components are realized using gate circuits.

In the first part of the design process we use trace theory to reason about all possible sequences of events. Components are specified by regular-expression-like programs, called commands, whose semantics is based on directed trace structures. We formalize the concepts of speed-independent and delay-insensitive circuits in the context of a network of basic components.

In the second part we use switching theory for the analysis of gate circuits. Three different delay models are discussed: the feedback-delay, the gate-delay, and the gate-and-wire-delay model. The last two models correspond to speed-independent and delay-insensitive circuits, respectively. We point out that networks of components are commonly operated in the 'input-output mode' (where inputs may change as soon as outputs have responded to a previous input change), whereas gate circuits are usually operated in the 'fundamental mode' (where the entire gate circuit must stabilize before another input change is permitted).

We note that delay-insensitive gate circuits are unlikely to exist for most basic components. For this reason, it is important that analysis and design methods are developed using bounded-delay models.

*Presented at the Seventh International Conference FUNDAMENTALS OF COMPUTATION THEORY FCT '89, Szeged, Hungary, August 21-25, 1989.

1 Introduction

In recent years a number of important results have been obtained in the area of asynchronous circuits. The purpose of this paper is to describe some of these key developments and to refer to others which, for lack of space, cannot be discussed properly here.

Before we present the new results, we briefly emphasize the increasing importance of asynchronous circuits in the rapidly changing world of computer technology. As computer systems become more and more distributed, it is more difficult to achieve proper communication and synchronization among all the parts in such systems. Each of these parts is usually an independently clocked (i.e. synchronous) system. The synchronization of such systems involves many timing problems, some of which —like the metastability problem[6] — are of a fundamental nature. It is believed that proper design techniques for asynchronous circuits can alleviate these problems considerably.

For many years asynchronous circuits have been studied using Boolean algebra as the main formalism. These studies started with Huffman[13] and Muller and Bartky[19]; in the latter work the name *speed-independent circuit* was coined. Several different models[2, 3, 11, 23] were applied in order to describe and verify circuit behaviour as accurately as possible. It was only recently that a unifying theory was found in which the differences and similarities among these models could be explained[5].

A somewhat different approach to the design of asynchronous circuits was advocated by Molnar et al. in the Macro Modules project[7] from which the term *delay-insensitive circuit* evolved and, more recently, by Seitz[25] who coined the term *self-timed system*. Ideas expressed by Molnar and Seitz have influenced researchers at Eindhoven University of Technology[10, 20, 21, 27, 29] where a formalism, called *trace theory*, was developed for the design of such circuits. A similar formalism was used recently by Dill[8] for automatic verification of speed-independent circuits.

Martin[14, 16] uses the language of *Communicating Sequential Processes* (CSP)[12] to specify the behaviours of components to be designed. Such a CSP specification can then be compiled into a self-timed circuit. Many interesting circuits have been designed in this way, culminating with a fast asynchronous microprocessor[15]. Techniques similar to Martin's were also applied at Philips Research[1] where unexpectedly good results have been obtained.

Another demonstration of the usefulness of asynchronous circuits was given by Ivan Sutherland in his 1988 Turing Award lecture[28], where he shows how special types of asynchronous circuits, called *micropipelines*, can be used conveniently in the design of many fast processing components.

The design of asynchronous circuits is an attractive area of research, in particular because it lies on the boundary of theory and practice. On the

one hand it contains simple and elegant mathematics, from formal language theory to semantics. On the other hand it is practical: many circuits have been used in actual designs and exhibit an unexpectedly good performance and robustness. Furthermore, such circuits are particularly well-suited for implementing parallel computations.

Although asynchronous circuits have been studied for many years now, the new approaches and major breakthroughs make us believe that this field is still very young and that more results are to be expected in the near future. Some difficult problems, however, still remain. We discuss some of these problems in the next sections.

2 The Producer and Consumer Paradigm

To illustrate the design of asynchronous circuits, we discuss a simple example starting with a behavioural specification and ending with a gate-level implementation. The necessary terminology and notation will be developed along the way. The producer-consumer setting of the example is due to Dijkstra[9]; the final circuit is a special case of a micropipeline[28].

Consider a 'producer' that outputs data items to be stored in a buffer and a 'consumer' that removes such items from the same buffer. Let a and b denote the production and consumption of data items, respectively. The producer and consumer act independently of each other; consequently, together they might generate any sequence of a's and b's. This may lead to unacceptable situations, however: the producer may cause an overflow of the buffer —which is assumed to be finite— and the consumer may cause an underflow. Consequently we need to design a controller which ensures that no items are produced if the buffer is full and none are consumed if the buffer is empty.

To keep the example simple, assume that the buffer has 2 places. Then the set of allowed sequences of productions and consumptions is the language accepted by the (incomplete) finite automaton defined by the state graph of Figure 1, where the initial state (corresponding to the empty buffer) is designated by an incoming arrow, and all the states are accepting states. Examples of allowed sequences are: ϵ (the empty trace), a, aa, ab, aab, $aabb$, etc.

Figure 1: State graph for buffer controller.

In order to ensure that only allowed sequences occur, the controller of

Figure 2 will send an 'acknowledge' signal to the producer and one to the consumer. Consider first the producer side and assume that the buffer is not full. The producer supplies input a notifying the controller that an item is being stored. After some delay, the controller will respond with an output p, informing the producer that another item can be stored. For convenience, signals that are inputs (respectively outputs) to the controller will be identified by ? (respectively !).

Figure 2: Buffer controller.

When the consumer and the controller are considered in isolation, the communication protocol between the two consists of an alternation of input a and output p, starting with a. Thus, all allowed behaviours are defined by the state graph of Figure 3(a). Similarly, the protocol between the controller and consumer is as shown in Figure 3(b). Initially the buffer is empty. When it becomes non-empty, the controller informs the consumer of this fact by sending output q. The consumer then removes an item while sending input b to the controller. This then repeats.

Figure 3: (a) Producer interface (b) Consumer interface.

In the following we give a formal specification of the buffer controller using trace theory; the material from here to Section 7 is based on [10, 20, 27, 29]. Finite sequences of symbols are called *traces* and sets of such sequences, together with the indication which symbols are inputs and which are outputs, are described by *directed trace structures*. Formally, a directed trace structure is a triple $< A, B, X >$, where A is the *input alphabet*, B is the *output alphabet*, and X is a set of traces constructed from symbols in $A \cup B$. A trace structure is called *regular* when its trace set is a regular set. Regular directed trace structures can be represented by expressions called *commands*, which are similar in many ways to regular expressions. Commands are defined inductively.

The atomic commands ϵ, $a?$, $a!$, and $!a?$ represent the atomic trace structures $< \emptyset, \emptyset, \{\epsilon\} >$, $< \{a\}, \emptyset, \{a\} >$, $< \emptyset, \{a\}, \{a\} >$, and $< \{a\}, \{a\}, \{a\} >$, respectively. For commands E, $E0$, and $E1$, the expressions $E0; E1$ (concatenation), $E0|E1$ (union), $[E]$ (repetition), and $\mathbf{pref}E$ (prefix-closure) are also commands. Let $\mathbf{i}E, \mathbf{o}E$, and $\mathbf{t}E$ denote the input alphabet, output alphabet, and trace set of the directed trace structure represented by command E. The directed trace structures represented by $E0;E1$, $E0\,|\,E1$, $[E]$, and $\mathbf{pref}E$ are defined by

$$\begin{aligned} E0; E1 &= < \mathbf{i}E0 \cup \mathbf{i}E1, \mathbf{o}E0 \cup \mathbf{o}E1, (\mathbf{t}E0)(\mathbf{t}E1) >, \\ E0 \mid E1 &= < \mathbf{i}E0 \cup \mathbf{i}E1, \mathbf{o}E0 \cup \mathbf{o}E1, \mathbf{t}E0 \cup \mathbf{t}E1 >, \\ [E] &= < \mathbf{i}E, \mathbf{o}E, (\mathbf{t}E)^* >, \text{ and} \\ \mathbf{pref}E &= < \mathbf{i}E, \mathbf{o}E, \{t_0 \mid (\exists t_1 :: t_0 t_1 \in \mathbf{t}E)\} >, \end{aligned}$$

where concatenation of sets is denoted by juxtaposition and * denotes Kleene's closure. (Here, we use the same notation for the command and the language defined by the command.) With the above definitions, the communication behaviours between producer and controller and between consumer and controller can be described by $\mathbf{pref}[a?;p!]$ and $\mathbf{pref}[q!;b?]$ respectively.

3 Parallel Composition and Synchronization

The complete communication behaviour of the controller of Figure 2 is specified by a proper synchronization of the two communication protocols; the overall protocol must ensure that the number of items contained in the buffer is always at most two and at least zero. In order to describe this proper cooperation between the two sides of the controller, we introduce a new operation on directed trace structures called *weaving*.

Formally, the weave $E0||E1$ of two directed trace structures represented by the commands $E0$ and $E1$ is defined by

$$\begin{aligned} E0||E1 = &< \mathbf{i}E0 \cup \mathbf{i}E1, \mathbf{o}E0 \cup \mathbf{o}E1, \\ &\{t \in (\mathbf{a}E0 \cup \mathbf{a}E1)^* \mid t{\downarrow}\mathbf{a}E0 \in \mathbf{t}E0 \wedge t{\downarrow}\mathbf{a}E1 \in \mathbf{t}E1\} > . \end{aligned}$$

Here, $\mathbf{a}E = \mathbf{i}E \cup \mathbf{o}E$ and $t{\downarrow}B$ denotes the projection of trace t on alphabet B, i.e. the trace from which all symbols not in B have been deleted. Informally, a weave of two specifications represents all behaviours that are in accordance with each of the two specifications.

As an example, consider the two commands $E0 = \mathbf{pref}[a?;c!]$ and $E1 = \mathbf{pref}[b?;c!]$. According to the above definitions of weaving we have

$$\mathbf{i}(E0||E1) = \{a, b\},\ \mathbf{o}(E0||E1) = \{c\},\ \text{and}$$

$$\mathbf{t}(E0 \| E1) = \{\epsilon, a, b, ab, ba, abc, bac, ..\}.$$

This directed trace structure can be represented also by $\mathbf{pref}[a?\|b?;c!]$.

Notice that, in a weave, common symbols must match. One could also say that weaving expresses 'parallel co-operation with synchronization on common symbols.' There are two special cases of weaving $E0$ and $E1$: if $\mathbf{a}E0 \cap \mathbf{a}E1 = \emptyset$, weaving amounts to interleaving or shuffle; if $\mathbf{a}E0 = \mathbf{a}E1$, weaving amounts to intersection.

Returning to the buffer, we give a specification for the communication behaviour of the controller using weaving and projection. For this purpose we introduce a so-called *internal symbol* $!x?$ in the two communication protocols to achieve proper synchronization. This internal symbol is 'projected away' after weaving. The complete communication behaviour of the controller is given by

$$E = (\mathbf{pref}[a?;!x?;p!] \parallel \mathbf{pref}[!x?;q!;b?]) \downarrow \{a,p,b,q\}.$$

Because of the synchronization on the common (internal) symbol $!x?$, there are always at most two and at least zero items in the buffer. To see this, notice that, because of the first command in the weave, $0 \leq \#_a t - \#_x t \leq 1$ for each trace t in $\mathbf{t}E$, where $\#_a t$ denotes the number of a's in t. Similarly, because of the second command in the weave, $0 \leq \#_x t - \#_b t \leq 1$. Consequently, we have $0 \leq \#_a t - \#_b t \leq 2$. Moreover, each trace t with this property is also contained in $\mathbf{t}E$.

A command equivalent to E is the following:

$$E1 \;=\; \mathbf{pref}(a?;[(p!;a?)\|(q!;b?)]).$$

Both commands define the same trace structure. From this last command it is readily verified that the communication behaviour specified by E can be represented also by the state graph of Figure 4.

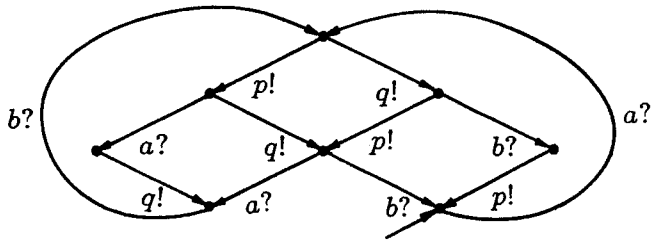


Figure 4: State graph for E.

The command $E1$ does not generalize to the case where the buffer has $n > 2$ places. The state graph of Figure 4 does generalize, but its size is

exponential in n. The command E generalizes easily. For example, for $n = 4$ the command becomes

$$(\ \mathbf{pref}[a?; !x1?; p!] \parallel \mathbf{pref}[!x1?; !x2?]$$
$$\parallel\ \mathbf{pref}[!x2?; !x3?] \parallel \mathbf{pref}[!x3?; q!; b?]) \downarrow \{a, p, b, q\}.$$

One can show that $0 \leq \#_a t - \#_b t \leq 4$ for each trace t in the above command [27].

The length of the generalized version of the command E is linear in n. This illustrates that commands may be preferred to state graphs in case parallel operation and synchronization are involved.

4 Specification and Implementation

Thus far, we have specified the communication behaviour of the buffer controller by means of the rather abstract notion of a directed trace structure, which can be represented by a command. Such a specification can be interpreted not only in a formal mechanistic way, but also in physical terms like voltage transitions on wires. We explain these interpretations by means of the specification for the so-called *(Muller) C-element* (named after Muller[19]).

Formally, a C-element is specified by the command $\mathbf{pref}[a?\|b?; c!]$. It represents a basic component with three terminals: inputs a and b, and output c. Its schematic is shown in Figure 5. Initially, the environment for this

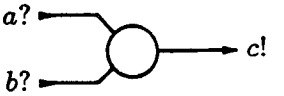
C-element	$\mathbf{pref}[a?\|\|b?; c!]$	
WIRE	$\mathbf{pref}\ [a?; b!]$	
IWIRE	$\mathbf{pref}\ [b!; a?]$	

Figure 5: C-element, WIRE, and IWIRE.

component produces communication actions (at terminals) a and b; then the component will respond with a communication action (at terminal) c. Only after c has been received may the environment produce the next communication actions a and b, after which the component will respond with c again, etc. We call this mode of operation —where outputs may be generated only after certain inputs have occurred and where next inputs may be generated only after certain outputs have occurred— the *input-output mode* of operation.

In a physical interpretation of the C-element, the symbols a, b, and c stand for voltage transitions at the corresponding terminals. A voltage transition can be high-going or low-going; both transitions are denoted by the same symbol. Input transitions are caused by the environment and output transitions are caused by the circuit.

With this physical interpretation in mind we can construct the state graph of Figure 6 for the behaviour of the C-element; the states in this graph are represented by the voltage levels at terminals a, b, and c respectively. Initially all voltages levels are 0. For convenience, n-tuples like a, b, c and $0, 0, 0$ are written as abc and 000, etc. The unstable states are represented by dashed circles.

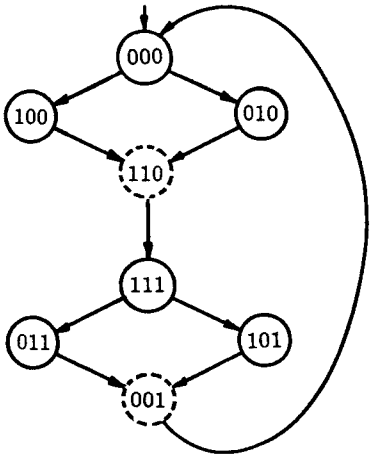


Figure 6: State graph for C-element.

In Figure 5 specifications of two other basic components, viz., the WIRE and the IWIRE, are given as well. The WIRE component has two terminals. It first receives input a and then responds with output b. According to the input-output mode of operation, the environment may produce the next input a only after it has received the output b, after which the WIRE will respond with output b again, etc. The IWIRE can be seen as an initialized WIRE: it starts by producing an output; after this, its behaviour is the same as that of the WIRE. The IWIRE is denoted by an open arrowhead in the schematic.

5 Decomposition

Given a specification of the communication behaviour of a component, like command E for the controller, we would like to 'decompose' this component into a network of some 'basic' components. In other words, we would like to

find a network of basic components that produces the outputs as specified in E, if the environment produces the inputs as specified in E.

We first illustrate the concept of decomposition by means of the network of Figure 7. This network consists of a connection of two WIRE components,

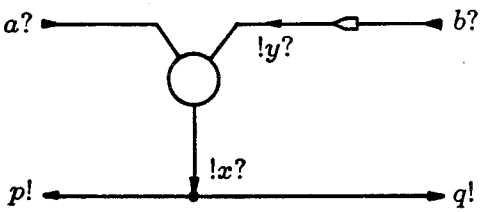


Figure 7: A network of basic components.

one IWIRE, and one C-element. Their respective specifications are $E_1 = \mathbf{pref}[x?;p!]$, $E_2 = \mathbf{pref}[x?;q!]$, $E_3 = \mathbf{pref}[y!;b?]$, and $E_4 = \mathbf{pref}[a?\|y?;x!]$. We show that the controller specified by

$$E = (\mathbf{pref}[a?;!x?;p!] \parallel \mathbf{pref}[!x?;q!;b?]) \downarrow \{a,p,q,b\}$$

can be decomposed into E_1, E_2, E_3, and E_4. This property is expressed by $E \rightarrow (E_1, E_2, E_3, E_4)$, where the network of the components E_1, E_2, E_3 and E_4 is denoted by (E_1, E_2, E_3, E_4).

In order to take the environment of the network into account as well, we take the *reflection* of E in which we interchange the role of component and environment. More formally, the reflection of a directed trace structure represented by command E, is denoted by $\overline{E}$ and defines the directed trace structure $\overline{E} = <\mathbf{o}E, \mathbf{i}E, \mathbf{t}E>$, where the inputs and outputs are interchanged. In our example, $\overline{E}$ specifies when its outputs a and b are produced; these form the inputs of the network (E_1, E_2, E_3, E_4). Instead of considering the network (E_1, E_2, E_3, E_4) and its environment as specified in E, we consider the network $(E_0, E_1, E_2, E_3, E_4)$ from now on, where $E_0 = \overline{E}$.

Formally, in order to prove that the network of Figure 7 behaves as specified in E, we need to demonstrate that four conditions hold for the network $(E_0, E_1, E_2, E_3, E_4)$. The first condition requires that there be no dangling inputs or outputs, i.e. that every input be connected to an output and vice versa. In formula:

$$(\cup i : 0 \leq i < 5 : \mathbf{o}E_i) = (\cup i : 0 \leq i < 5 : \mathbf{i}E_i). \qquad (1)$$

If (1) holds, we say that the network $(E_0, E_1, E_2, E_3, E_4)$ is *closed.*

The second condition is that no outputs of distinct components are connected to each other. In formula

$$\mathbf{o}E_i \cap \mathbf{o}E_j = \emptyset \quad \text{for } 0 \leq i, j < 5 \wedge i \neq j. \tag{2}$$

When (2) holds, we say that the network is *free of output interference.* If (1) and (2) hold, then every symbol is an output of only one component in the network.

Conditions (1) and (2) are conditions on the structure of the network. They are formulated in terms of the alphabets of the directed trace structures. The next two conditions are behavioural conditions; they are phrased in terms of the trace sets and the alphabets.

The third condition prescribes that the input-output mode of operation may not be violated for any component in the network. We can simulate the network by generating traces of symbols, representing joint behaviours of the components in the network. Formally, we construct the trace set X of all joint behaviours as follows. Initially, $X = \{\epsilon\}$. Choose a trace t, symbol z, and index $i, 0 \leq i < 5$, such that $t \in X \wedge z \in \mathbf{o}E_i \wedge tz \downarrow \mathbf{a}E_i \in \mathbf{t}E_i$ holds (i.e. after joint behaviour t, component E_i can produce output z). If for all $j, 0 \leq j < 5$ we have $tz \downarrow \mathbf{a}E_j \in \mathbf{t}E_j$, (component j can accept z, i.e. its input-output mode of operation is not violated), then we add tz to X. Otherwise, we stop the simulation and say that the network has *computation interference.* Our third condition is

$$\text{The network is free of computation interference.} \tag{3}$$

Testing for computation interference can be done by an algorithm involving a finite state graph[8, 10]. One can verify that (3) is satisfied for the network $(E_0, E_1, E_2, E_3, E_4)$. The trace set X that can be generated for this network can be represented by

$$X = \mathbf{t}(\mathbf{pref}[!a?; !x?; !p?] \parallel \mathbf{pref}[!y?; !x?; !q?; !b?]).$$

The only difference between X and $\mathbf{t}E$ is the symbol y which we introduced in the decomposition as the output of the IWIRE and input of the C-element.

The fourth condition is that every trace of the component specified (here E) may also occur in the simulation we described above (excluding symbols not in $\mathbf{a}E$) and that only such traces may occur. In formula:

$$X \downarrow \mathbf{a}E = \mathbf{t}E. \tag{4}$$

If (4) is satisfied we say that the network behaves as specified. Condition (4) is satisfied by the network of Figure 7 as well.

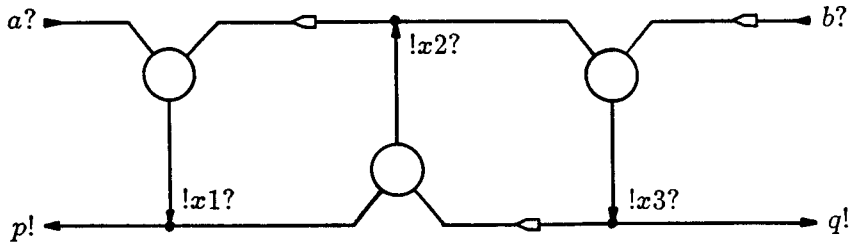


Figure 8: Decomposition for 4-place buffer controller.

The decomposition of Figure 7 generalizes to a decomposition for the controller of the n-place buffer. The network for this decomposition is given in Figure 8 for $n = 4$. Notice that the synchronization on common symbols is realized by the C-elements.

The formalization of decomposition as given above is taken from [10]. The verification of the proof obligations mentioned above can be automated. Dill has designed a verifier that checks whether conditions (1) through (3) hold [8]. Such a verification method, however, is proportional to the number of states in the global state graph, which can grow as the product of the numbers of states of the components. For this reason, it is essential that theorems be developed that allow for a more efficient design or verification of a decomposition.

6 DI Decomposition

In the previous section we gave a formal definition of decomposition in terms of trace structures. The physical interpretation of decomposition is intended to correspond to the realization of a circuit by a network of sub-circuits. These sub-circuits may have arbitrary, non-negative response times. The communications between the sub-circuits, however, are assumed to be instantaneous. Thus, a circuit obtained by means of decomposition can be called a *speed-independent circuit*, i.e. its correctness is independent of any delay in the response times of the components.

In practice, the sub-circuits are connected to each other by means of wires, which may have unspecified delays. Such delays may affect the correctness of the circuit. If the correctness of the circuit is independent of any delays in the response times of components *and* connection wires, then we call such a circuit a *delay-insensitive circuit.*

While a speed-independent circuit is formally described by means of a decomposition, a delay-insensitive circuit is formally described by means of a *DI decomposition.* A DI decomposition is a decomposition in which all connection wires between the components are taken into account. Formally, these connection wires are represented by WIRE components and connect

components with each other through an intermediate boundary as exemplified in Figure 9.

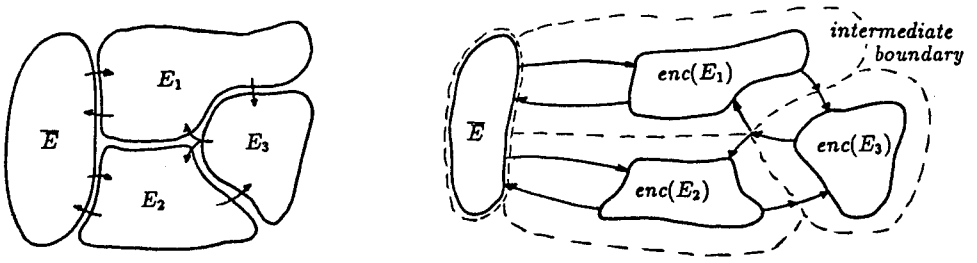


Figure 9: (a) Decomposition. (b) DI Decomposition.

We give a brief description of a delay-insensitive circuit. For more details the reader is referred to [10]. First, we define the *enclosure* $enc(E_1)$, i.e. the component enclosed by the intermediate boundary, by renaming the symbols in the command E_1 to their 'localized' versions. The collection of WIRE components connecting the enclosure $enc(E_1)$ with its intermediate boundary is denoted by $Wires(E_1)$. E_2, E_3, and E_4 are treated similarly. We say that the components E_1, E_2, E_3, and E_4 form a DI decomposition of component E, denoted by $E \stackrel{DI}{\rightarrow} (E_1, E_2, E_3, E_4)$ if and only if

$$E \rightarrow (i : 0 \leq i < 5 : enc(E_i), Wires(E_i)\,). \tag{5}$$

In general, DI decompositions are more difficult to derive and verify than decompositions because of all the (connection) WIRE components. The two decompositions are equivalent, however, if all the constituent components are so-called DI components. A component E is called a DI component, if

$$E \rightarrow (enc(E), Wires(E)).$$

This property formalizes that the communication behaviour between component and environment is insensitive to wire delays. Verification of the DI property reduces to verifying that the network $(\overline{E}, Wires(E), enc(E))$ is free of computation interference. The basic components C-element, WIRE, and IWIRE, for example, are DI components.

Since all basic components of the decomposition of Figure 7 are DI components, this decomposition is a DI decomposition, i.e. (5) holds. Accordingly, the circuit of Figure 7 represents not only a speed-independent circuit but also a delay-insensitive circuit. The same reasoning holds for Figure 8.

The idea of formalizing delay-insensitivity using a characterization of a DI component originates from Molnar[18]. Udding was the first to give a rigorous formulation of the DI property in terms of directed trace structures[29].

Shannon showed that any switching function can be realized by a gate circuit with only a finite number of gate types[26]. Similarly, we can ask ourselves 'Can any DI component be decomposed into a network of components chosen from a finite basis of DI components?' In [10] it is shown that, indeed, any regular DI component can be so decomposed. Consequently, such a decomposition is also a DI decomposition. The C-element, WIRE, and IWIRE component are members of such a basis. Other components are, for example, the XOR (or MERGE) component specified by **pref**$[(a?|b?); c!]$, the TOGGLE component specified by **pref**$[a?; b!; a?; c!]$, and an arbiter-like component.

If a component is specified by a command satisfying a certain syntax, then its decomposition can be described as a syntax-directed translation into a network of basic (DI) components[10]. Another attractive property of this translation is that the number of basic components in the final network is proportional to the length of the command. We also mention, however, that the decompositions obtained thus may not be optimal.

7 Realizations of Components by Gate Circuits

Having decomposed a component to be designed into a network of basic components such as C-elements and WIREs, one is faced with the problem of realizing the basic components. Here, we assume that this is to be done using logic gates; space limitations prevent us from discussing other types of realizations, such as those based on the commonly used MOS technology[30].

We introduce a number of ideas related to the design of asynchronous gate circuits by using the example of the C-element. The input-output behaviour of the C-element of Figure 5 has been described by the state graph of Figure 6. We assume that the inputs a and b can only change one at a time. The following illustrates a frequently used approach to gate circuit design. Construct a combinational gate circuit with inputs a, b, and c, and output C. The output C gives the 'excitation' of c, i.e. the next value that the (sequential) circuit output c should assume, if the present values of the inputs and the output are given by a, b, and c. From Figure 6 we observe that the output c should become 1 if $a = 1$ and $b = 1$. Once the output c becomes 1, it should remain 1 as long as $a = 1$ or $b = 1$. Thus, we have $C = ab + (a + b)c = ab + ac + bc$, where ab denotes the AND function and $a + b$ denotes the OR function of a and b. A gate circuit corresponding to this expression is shown in Figure 10, where the rectangle between C and c represents a delay. The presence of such a delay is implicitly assumed when we talk about the present value c of the

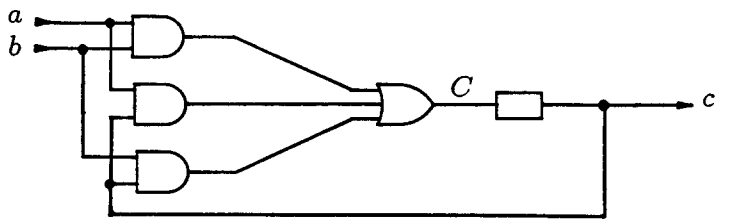


Figure 10: Gate circuit for C-element.

output and the excitation C to which the output is tending to change. Since C is assumed to be a Boolean function of a, b, and c, the value of C is computed from those of a, b, and c without delay.

Our design of the buffer controller began with a high-level specification of the controller component that led to the decomposition of the component in terms of some basic components. Furthermore, this decomposition has the important property that the network behaviour is independent of the delays in basic components and wires. Thus, the decomposition is delay-insensitive under the assumption that the environment co-operates, i.e. that the input-output mode of operation is used. A natural question now arises: Can each basic component be realized by a delay-insensitive *gate* circuit? In particular, is the circuit in Figure 10 for the C-element delay-insensitive? We consider such questions in the next two sections.

8 Fundamental Mode versus Input-Output Mode

Classical switching theory[17] assumes that gate circuits operate in *fundamental mode*. This means that the environment of the gate circuit co-operates in such a way that it produces a next input only after the entire circuit has stabilized. This is a much more restricted environment than the one in the input-output mode of operation, where an input is allowed to change as soon as the output changes, but possibly before all the gates in the circuit have had the chance to stabilize.

For simplicity, we first consider the question whether the circuit of Figure 10 is speed-independent. In other words, we assume that each gate has a delay and that wires have no delays. This model is also called the *gate-delay model*. The gate-delay model for the circuit of Figure 10 is shown in Figure 11. This model is more realistic than the one of Figure 10, where there is only one delay before the output c. Suppose that the circuit of Figure 11 is started in state $ab = 00, cdef = 0000$. When a changes to 1, the excitation remains $CDEF = 0000$, i.e. the circuit is stable and no further changes take place. Next, suppose b changes to 1, i.e. we reach $ab = 11, cdef = 0000$. Now

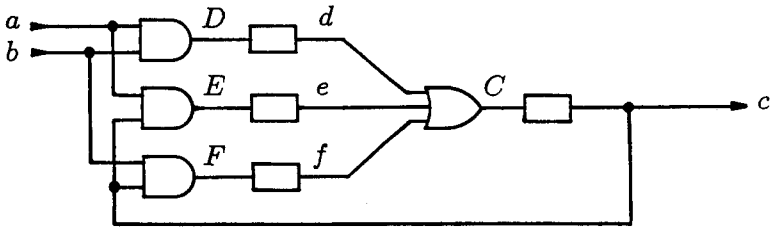


Figure 11: Gate-delay model.

$D = 1$, and the other excitations remain 0. After some time, d becomes 1 and changes C to 1. Eventually, c, e, and f also become 1 and the circuit stabilizes in $ab = 11, cdef = 1111$. Altogether, we have verified that, starting with $ab = 10$ and $cdef = 0000$, the change to $ab = 11$ results reliably in the state $cdef = 1111$, when fundamental mode is assumed.

Now consider the same transitions from the stable state $ab = 00, cdef = 0000$ when ab becomes 10 and then 11 in the input-output mode. The following sequence of events is possible, where underlined entries represent unstable gates:

ab	$cdef$	
00	0000	stable initial state
10	0000	input a changes, state is still stable
11	0$\underline{0}$00	input b changes
11	$\underline{0}$100	output d changes
11	11$\underline{0}\underline{0}$	output c changes

In the input-output mode, the environment may change input b again now. Thus, the following is possible:

10	1$\underline{1}$ $\underline{0}$0	input b changes
10	$\underline{1}$0 $\underline{0}$0	output d changes (before output e)
10	00 00	output c changes (before output e)

The final state reached in this transition is the stable state $ab = 10, cdef = 0000$. Consider the signals a, b, and c only; we have just shown above that the following trace is possible: $t = abcbc$. This is not in accordance with the C-element specification, which requires that all the traces must be in the trace set of $\mathbf{pref}[a?\|b?; c!]$.

Altogether, we have shown that a circuit behaving properly in fundamental mode may not behave properly in input-output mode. On the other hand, one can view fundamental mode operation as an input-output mode operation with 'slowly' changing inputs. If all gate outputs have become stable, then

the circuit outputs, being outputs of some gates, have also reached their new values. Hence, operating a circuit in fundamental mode does not violate any input-output mode principles.

9 Fundamental Mode Analysis

We have seen in the last section that fundamental mode operation may differ from input-output mode operation. However, we do not reject the fundamental mode approach for two reasons. First, if it is impossible to realize a circuit specification to operate correctly in fundamental mode, then it is certainly impossible to do this in input-output mode. Therefore, fundamental mode realizability is a necessary condition for input-output mode realizability. Second, very little work has been done on input-output mode analysis, whereas much is known about the fundamental mode approach. In this section we briefly summarize the known results concerning fundamental mode analysis, including some very recent findings.

The first question that arises when one is choosing a model for a gate circuit is what assumptions are to be made about the presence of delays in the circuit. Three different models have been used in the past. The first one is the *feedback-delay model*, where one chooses a set of wires in the circuit with the property that cutting all these wires removes all the loops in the circuit —thus making the resulting circuit a combinational one. One then associates a delay with each wire in this set. An example of such a model is the circuit of Figure 10, where only one delay is assumed. This model was introduced by Huffman[13].

The second model is the *gate-delay model* in which a delay is associated with each gate. This corresponds to the concept of speed-independence described in Section 5 for a decomposition of a component into a network of basic components. This model was used by Muller and Bartky[19]. An example of this approach is the circuit of Figure 11.

The third model, the *gate-and-wire-delay model*, corresponds to the concept of delay-insensitivity described in Section 6 for DI decomposition. Such a model was used implicitly or explicitly by many authors; see, for example, [3]. To illustrate this model, consider Figure 11; here one would have to add a delay in the wire from input a to the top AND gate and also one from input a to the middle AND gate, etc. Altogether, ten additional wire delays have to be introduced.

Having selected a delay model for a given circuit, we can associate a state variable with each delay and find the excitation function for that variable, i.e. the Boolean function that specifies the value to which that variable is tending to change. For example, for the single state variable c in Figure 10,

the excitation is $C = ab + ac + bc$. For Figure 11, we have four excitation functions: $D = ab$, $E = ac$, $F = bc$, and $C = d + e + f$.

In order to cover all three delay models, we will refer to a set of state variables and their excitation functions as a *network*. An *internal state* of a network is a tuple of binary values assigned to the state variable tuple, say y, and an *input state* is a tuple of binary values assigned to the input variable tuple, say x. For example, for the network of Figure 11 we have $x = (a, b)$ and $y = (c, d, e, f)$. A *total state* is a pair (input state, internal state). A state variable is *stable* in a given total state if its value in that state is equal to its excitation in that state. A total state is stable if all of its state variables are stable. In fundamental mode analysis we start with a stable total state $(x0, y0)$ of a network and then change the input to $x1$ and keep it at that value until the circuit 'has had a chance to stabilize.' Only then is the input allowed to change again. Of course, not all such transitions result in a single stable state. If more than one stable state can be reached or if an oscillation occurs, the behaviour is considered improper.

In general, the new state $(x1, y0)$ is unstable. If there is only one variable unstable, then eventually that variable must change, and a unique next total state is reached. If two variables are unstable, either can change first, or both can change at the same time. Thus, there are three possible next states. The situation where two or more variables are unstable is called a *race*. A commonly used *race model* —dating back to Huffman[13], but formalized by Brzozowski and Yoeli[3]— assumes that, in any unstable state, any subset of the set of unstable variables may 'win the race,' i.e. change to its corresponding excitation state. This model has been called the GMW (general multiple winner) race model.

It turns out that, if one uses the GMW race model, then each of the three delay models (feedback, gate, gate-and-wire) may yield different results[5]. The most accurate (and realistic) model of the three is the gate-and-wire-delay model. Unfortunately, the computation time for this model is exponential in the number of state variables. Recently, however, it has been shown [4] that the results for the gate-and-wire-delay model can be obtained also by an efficient method called *ternary simulation* introduced by Eichelberger[11]. Moreover, it was shown in [5] that, when a different race model —the so-called XMW model— is used, all three delay models yield the same results as ternary simulation. Thus, in the XMW race model one is permitted to use the feedback-delay model without losing any information. The XMW model uses ternary algebra based on the three values 0, 1, and $\times$, where the last value corresponds to an 'uncertain' signal.

While the XMW analysis (in any delay model) or, equivalently, the GMW analysis in the gate-and-wire-delay model give useful results, these results are frequently pessimistic in the sense that they include timing problems which

are very unlikely to occur in practice. These analysis models are indeed so pessimistic that only very few sequential circuit behaviours can be realized in delay-insensitive fashion[23]. For example, Seger has shown that there does not exist a delay-insensitive gate circuit realizing a modulo-2 counter.

The situation is even worse for the input-output mode operation. We conjecture, that such commonly used circuits as the set-reset latch and the C-element do not have delay-insensitive gate circuit implementations. In fact, it appears that no non-trivial sequential behaviours have such realizations.

10 Bounded-Delay Models

The results mentioned in the previous section are rather discouraging because the basic components needed for delay-insensitive decomposition cannot be designed in delay-insensitive fashion from gates. What then is the solution to this dilemma? The practical answer is that we have to make some assumptions about the sizes of delays in circuit elements and wires, i.e. we are led to some type of *bounded-delay model.* Such a model has been used informally for many years. See, for example, [22]. A simple example of such an approach is the following. First, design a gate circuit using Huffman's feedback variable approach (as illustrated by the example in Figure 10). Then introduce a sufficiently large delay in the output to make sure that all the gates and wires in the circuit stabilize before the new output value reaches the output terminal. Such an approach will work if each delay has an upper bound.

We illustrate the bounded-delay approach with the circuit of Figure 12 for the C-element. The unlabeled rectangles and the thin ovals represent the gate

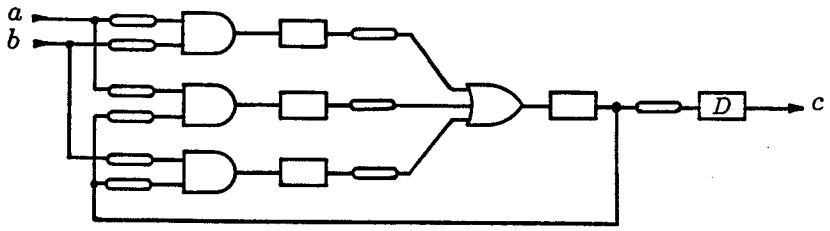


Figure 12: Bounded-delay model

and wire delays, respectively. The delay element labeled D is added by the designer to 'slow down' the output. Suppose we know that all wire delays are at most one time unit, and that all gate delays are at most two time units. One can verify that the circuit will behave properly in the input-output mode, if the output delay D is at least four time units. Notice that the presence of

the output delay forces the input-output mode operation to become identical to the fundamental mode operation of the circuit.

It is an open problem whether there exist general systematic design techniques for circuits operating in some appropriate bounded-delay model. In fact, the analysis of circuits under the bounded-delay assumption is far from trivial. Some new results have been obtained by Seger who has shown that bounded-delay analysis can be done efficiently[24].

11 Concluding Remarks

By means of a simple example we have illustrated some of the recent developments in the design of asynchronous circuits. Because of space limitations, we have not been able to discuss important results obtained by others. In particular, we would like to mention the recent developments made by A.J. Martin. The interested reader will find an extensive overview in[16].

References

[1] C. van Berkel, C. Niessen, M. Rem, R. Saeijs, VLSI Programming and Silicon Compilation: a Novel Approach from Philips Research, *Proceedings of IEEE International Conference on Computer Design 1988, (ICCD '88)*, 1988.

[2] J.A. Brzozowski and M. Yoeli, *Digital Networks*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[3] J.A. Brzozowski and M. Yoeli, On a Ternary Model of Gate Networks, *IEEE Transactions on Computers*, Vol. C-28, pp. 178-183, 1979.

[4] J.A. Brzozowski and C-J. Seger, A Characterization of Ternary Simulation of Gate Networks, *IEEE Transactions on Computers*, Vol. C-36, pp. 1318-1327, 1987.

[5] J.A. Brzozowski and C-J. Seger, A Unified Framework for Race Analysis of Asynchronous Networks, *Journal of the ACM*, Vol. 36, pp. 20-45, 1989.

[6] T.J. Chaney and C.E. Molnar, Anomalous Behavior of Synchronizer and Arbiter Circuits, *IEEE Transactions on Computers*, Vol. C-22, pp. 421-422, 1973.

[7] W.A. Clark and C.E. Molnar, Macromodular Computer Systems, in *Computers in Biomedical Research*, Vol. IV, (R. Stacy and B. Waxman, eds.), Academic Press, New York, 1974.

[8] D.L. Dill, Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits, in *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, (J. Allen and F. Leighton, eds.), MIT Press, pp. 51-68, 1988.

[9] E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, *Acta Informatica*, Vol. 1, pp. 115-138, 1971.

[10] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, CWI Tract 56, Centre for Mathematics and Computing Science, Amsterdam, 1989.

[11] E.B. Eichelberger, Hazard Detection in Combinational and Sequential Switching Circuits, *IBM Journal of Research and Development*, Vol. 9, pp. 90-99, 1965.

[12] C.A.R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, Vol. 21, pp. 666-677, 1978.

[13] D.A. Huffman, The Synthesis of Sequential Switching Circuits, in *Sequential Machines: Selected Papers*, (E.F. Moore ed.), Addison-Wesley, Reading Massachusetts, pp. 3-62, 1964, First appeared in the *J. Franklin Inst.*, Vol. 257, pp. 161-190, 1954.

[14] A. J. Martin, Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, *Distributed Computing*, Vol. 1, pp. 226-234, 1986.

[15] A. J. Martin et al., The Design of an Asynchronous Microprocessor, in *Advanced Research in VLSI, Proceedings of the Decennial Caltech Conference on VLSI*, (C.L. Seitz ed.), 1989.

[16] A. J. Martin, Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits, in *UT Year of Programming Institute on Concurrent Programming*, (C.A.R. Hoare ed.), Addison-Wesley, 1989.

[17] E.J. McCluskey, *Introduction to the Theory of Switching Circuits*, McGraw-Hill Book Company, New York, 1965.

[18] C.E. Molnar, T.P. Fang and F.U. Rosenberger, Synthesis of Delay-Insensitive Modules, in *Proceedings 1985, Chapel Hill Conference on VLSI*, (H. Fuchs ed.), Computer Science Press, pp.67-86, 1985.

[19] D. E. Muller and W.S. Bartky, A Theory of Asynchronous Circuits, *Proceedings of an International Symposium on the Theory of Switching*, Vol. 29 of the *Annals of the Computation Laboratory of Harvard University*, Harvard University Press, Cambridge, Mass., pp. 204-243, 1959.

[20] M. Rem, Concurrent Computations and VLSI Circuits, in *Control Flow and Data Flow: Concepts of Distributed Computing*, (M. Broy ed.), Springer-Verlag, pp. 399-437, 1985.

[21] M. Rem, Trace Theory and Systolic Computations, in *Proceedings PARLE, Parallel Architectures and Languages Europe*, Vol. 1, (J.W. de Bakker, A.J. Nijman and P.C. Treleaven eds.), Springer-Verlag, pp. 14-34, 1987.

[22] F. Rosenberger, C. Molnar, T. Chaney, and T-P. Fang, Q-modules: Internally Clocked Delay-Insensitive Modules, *IEEE Transactions on Computers*, Vol. 37, pp.1005-1018, 1988.

[23] C-J. Seger, *Models and Algorithms for Race Analysis in Asynchronous Circuits*, Ph. D. Thesis, Department of Computer Science, University of Waterloo, Research Report CS-88-22, 1988.

[24] C-J. Seger, The Complexity of Race Detection in VLSI Circuits, in *Advanced Research in VLSI, Proceedings of the Decennial Caltech Conference on VLSI*, (C.L. Seitz ed.), pp. 335-350, 1989.

[25] C.L. Seitz, System Timing, in *Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley, pp. 218-262, 1980.

[26] C. E. Shannon, A Symbolic Analysis of Relay and Switching Circuits, *Trans. AIEE*, pp. 731-723, 1938.

[27] J. L.A. van de Snepscheut, *Trace Theory and VLSI Design*, Lecture Notes in Computer Science 200, Springer-Verlag, 1985.

[28] I. E. Sutherland, *Micropipelines, The 1988 Turing Award Lecture*, to appear in CACM.

[29] J. T. Udding, A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems, *Distributed Computing*, Vol. 1, pp. 197-204, 1986.

[30] N. Weste and K. Eshragian, *Principles of CMOS VLSI Design A Systems Perspective*, Addison Wesley, 1985.