Math. Struct. in Comp. Science (2003), *vol.* 13, *pp.* 771–798. © 2003 Cambridge University Press
DOI: 10.1017/S0960129503004067 Printed in the United Kingdom

Concurrent computing machines and physical space-time

PHILIPPE MATHERAT[†] and MARC-THIERRY JAEKEL[‡]

[†]*Laboratoire Traitement et Communication de l'Information, ENST, 46 rue Barrault, F75013 Paris, France.*
[‡]*Laboratoire de Physique Théorique, ENS, 24 rue Lhomond, F75005 Paris, France.*

Received 8 January 2001; revised 30 October 2001

Concrete computing machines, either sequential or concurrent, rely on an intimate relationship between computation and time. We recall the general characteristic properties of physical time and of present realisations of computing systems. We emphasise the role of computing interferences, that is, the necessity of avoiding them in order to give a causal implementation to logical operations. We compare synchronous and asynchronous systems, and give a brief survey of some methods used to deal with computing interferences. Using a graphic representation, we show that synchronous and asynchronous circuits reflect the same opposition as the Newtonian and relativistic causal structures for physical space-time.

1. Introduction

Are concurrent computing machines equivalent to Turing machines? This question, which amounts to confronting the two fundamental notions of time and computation, may be treated in a purely mathematical framework. Practical consequences, however, cannot be independent of concrete realisations, that is concrete machines performing actual computations in physical time.

This remark may seem strange, if one aims to show theorems, which cannot depend on the physical properties of time or machines. But, even in a mathematical treatment of concurrent computation, one needs a representation of time. Usually, time is modelled as a real parameter, shared by all parts of the computation. Unfortunately, such a representation does not correspond to the observable time that can be obtained from physical systems like clocks, nor to the reference time that is defined by metrology, nor to the operational time that occurs in practical realisations of logic circuits. Without questioning the validity of proved theorems, difficulties may emerge when trying to find practical applications.

The distinction just made between abstract and concrete machines raises some related questions. When a machine M can be simulated by a program P running on another machine, how can one identify the concrete machine M and the program P? And if a machine cannot be simulated on another one, indicating some greater expressive power, is the latter due to a computational or a fundamental physical property? Before addressing

[†] Laboratoire du CNRS et de l'Ecole Nationale Supérieure des Télécommunications.
[‡] Laboratoire du CNRS, de l'Ecole Normale Supérieure et de l'Université Paris Sud.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

such questions, we must first consider the sequential machines that are currently realised. These machines, which we get from a constructor, are made of matter, and transform electric energy into heat, and which we communicate with through a keyboard and a screen, why do we need them? We may observe that all the features that make them concrete are inconveniences: we would prefer them lighter, smaller, less power consuming and less dissipating. Ideally, all these parameters would be equal to zero. In fact, we need these machines for their logical function – their ability to compute. But then, since this function is mathematically known, modelled and even simulated, why do we need a concrete machine, whose features are mainly inconveniences? The natural answer is that these machines compute faster than humans can, with just a pencil and paper. And yet, pencil and paper are already rudimentary elements of a concrete machine, using physical objects to remember different steps of computations. The interest in concrete machines comes from their intimate relation with physical time.

If computing machines go faster than humans, one must be confident in their action, as in most cases one is unable to check their output. Indeed, in very specific cases, one can verify the correctness of a result in much less time than is needed to obtain it. This is the case for instance of the prime factorisation of integers. But few concrete applications have this property. In most cases, one cannot check the result in much less time than the computation itself. If the result is important, and there is no other way to obtain it, we must have confidence in the machine.

What can give support for such confidence? Necessarily, the answer is reasoning, founded on the correct functioning of the machine at a given time on some particular computations, generalised to other times and other computations. A computing machine cannot be tested for all the computations it can do, at any time. Even for a finite machine, the number of possible computations increases exponentially with the memory size, and a memory of one hundred bits already allows a number of configurations that cannot be tested in less time than the age of the universe. To establish a reasoning scheme leading to confidence, one must:

— check that each elementary component effectively realises the function it has been designed for (physical validation).
— prove in a deductive way that the particular composition of these elementary components building the machine effectively leads to the global function used (logical validation).

The first condition is ensured by choices in implementation design and by tests made by the constructor. The second condition is obtained from a mathematical representation of the machine and from the logics of computation. These two steps of validation require good representations of all components at the physical level, and of the global machine at the logical level. If the confidence one can put in a machine relies on good models of both its physical and logical functioning, how could such a machine perform more than it has been designed for, more than our present theories can model? Even if the existence of a new type of calculus, still unknown today, can be envisaged, with machines performing this new type of calculus, how could one build such machines without having

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

good models for them? In such a case, one could not ensure the two validation steps, and one could not say that these machines operate correctly, nor be confident in their ouput.

One consequence is that realistic models are necessary, both of the computational structure and of the physical implementation of logical operators. Modelling is made easier when logical and physical constraints can be separated. This is the reason for developing sequential machines or synchronous concurrent machines. In that case, the logical validation of the machine can be made, whilst ignoring the implementation characteristics of its components. The latter will finally and mainly limit the performance of the machine through the value of the clock frequency. The machine can, equivalently, be simulated on another concrete machine with identical clock frequency, at the expense of slower performances. However, in the case of asynchronous concurrent machines, logical and physical constraints are more involved. Although machines built with asynchronous circuits are less widely used, much effort has been devoted to their understanding and modelling (Seitz 1980; Sutherland 1989; Martin 1990a; Ebergen 1991; Davis and Nowick 1995; Mallon *et al.* 1999). In fact, they may even appear as an unavoidable evolution of computing machines. On the one hand, clock-timed circuits are reaching limits where clock signal distribution consumes too many resources and progress in performance approaches saturation point. On the other hand, asynchronous circuits constitute the most general class of circuits, and thus allow one to express in the most general way the questions raised by the implementation of computation on physical systems, and the solutions that may be brought.

In this article, we shall be concerned with the relation of concrete computing machines with physical time. After recalling the general characteristic properties of physical time and of computing machines that are presently realised, we shall compare the solutions provided by synchronous and asynchronous systems to the implementation of logical operations. We shall show that they give different implementations of causal relations, reflecting in that way different causal structures for space-time.

2. Physical time

The notion of time may be seen to follow from two necessities. From a logical point of view, time can be considered as the concept that allows one to make a distinction between two different types of propositions: general and universal propositions (like mathematical ones) that are eternal, and particular propositions that are related to changing reality (like those describing physical systems). Moreover, time is also rendered necessary by the formulation of physics: time is the concept that allows one to give a formal expression to movement, and hence to the laws of physics.

The properties of time are in fact imposed by the functions that this notion must fulfill. From the logical side, time allows one to conceive of the same object by characterising it by its different states, these states being asssociated with the object at different times. A time parameter can then be used not only to index the different states characterising the same object, but also to organise the states of different objects into classes of simultaneity.

The relation of order that can be introduced on the time parameter allows one to define a relation of logical causality between the state transitions affecting different objects.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

However, in order to be realised physically, for instance on real machines, the causal relation between states cannot be independent of the real motions affecting physical systems. In particular, the simultaneity classes defined with the help of the time parameter must coincide with those that are associated with real events occurring in physical space, hence with the physical time.

The notion of physical time is intimately related to the laws of physics. After observing that pendulum oscillations are isochronous, Galileo Galilei was able to give a mathematical representation of motion induced by free fall, by relating the distance travelled to the elapsed time, the latter being understood as a universal reference for all motions. The existence of such a reference is made possible by the existence of physical laws governing all movements, and, in particular, by the existence of regular movements such as inertial motions.

This introduction of time leaves an important conventional part in the definition of a time reference, even if a natural choice is provided by motions that appear as most regular, like the Earth motion around the Sun. This leads us, in fact, to distinguish two types of time. Thus, Leibniz (Russell 1900), relying on logical arguments, could consider that space and time are mere relations between objects or events, which are fixed by an observer in a conventional way, thus building subjective space and time. Still, one is also bound to admit the existence of objective space and time, as the only way to understand how physical laws governing displacements of objects and time ordering of events can be formulated in a universal way, independently of the observer.

The formulation of the universal law of gravitation led Newton (Newton 1687) to fix the role played by time in physical laws, and to endow it with the mathematical representation that we still use nowadays: that of a real parameter that all physical quantities depend on. In fact, Newton introduced two different notions of time, which he distinguished both in their conception and in their usage. The first one, which he called 'absolute and mathematical', allowed him to write mathematical equations for the laws of mechanics and gravitation. The second notion, which he called 'common and sensible', allowed him to relate the motions of different physical systems, including clocks. Even if Newton emphasised the first notion, which he considered as representing absolute space and time, seeing clocks as systems to be improved in order to make them as close as possible to ideal space and time, he nevertheless made two distinct uses of these notions. The first, which he identified with the curvilinear coordinate on the planet's trajectory, he used as a mathematical tool to deal with infinitesimals of different orders. The second, which is the physical time as can be defined by Kepler's area law, he used as a measure of inertial motions, with which he compared planetary motions.

The theory of relativity (Einstein 1905) has led us to question the *a priori* and absolute character of physical space and time. According to relativity, the notion of time relies on clocks, the date of an event being defined by coincidence of this event with a tick delivered by a clock located at the same place. But in order to be defined throughout space, the notion of time also relies on the exchange of light signals, which are necessary to compare and synchronise the times shown by remote clocks. The universal and finite velocity of light propagation then leads to a definition of time simultaneity that depends on the observer's motion. In other words, time simultaneity is not given *a priori* but

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

results from a construction, or clock synchronisation. By exchanging light signals, on which time references provided by clocks are encoded, one can compare these references and synchronise clocks. Then, time allows one to construct space. By comparing the light signals received from several remote clocks, one can, by quadrangulation, determine positions both in time and space. This relativistic definition of time and space is rendered necessary as soon as a high precision must be attained. This is the case, for instance, when corrections linked to the finite velocity of light, or relativistic effects, must be taken into account (Landau and Lifschitz 1970; Wrinkler 1991). Hence, this relativistic definition is the one used in physics for high precision space-time measurements (Vessot 1991), and in metrology to define time and space standards (Quinn 1991) and to construct the space-time reference systems required by physics (Petit and Wolf 1994; Wolf and Petit 1995). It is also the one used in modern practical positioning systems at the surface of the Earth, such as GPS (Lewandowski and Thomas 1991; Leschiutta 1991). Finally, as will appear in the following, it is also the notion of time that is implicitly used by asynchronous communicating and computing systems (Lamport 1978).

The consequences of the theory of relativity on our conception of space and time have been remarkably discussed at the logical level by Russell (Russell 1925; Russell 1927). Our representation in terms of permanent material structures located in space and evolving according to a unique external time, must be replaced by one in terms of events that are located both in space and time. This conception of space-time not only affects the formulation of modern theories in a fundamental way (Jaekel and Reynaud 1998), but also underlies present applications in physics and metrology (Guinot 1997).

When refering to physical time, simultaneity classes can no longer be defined *a priori*, and we must rely on a physical implementation by means of propagating light signals. This constructive character of time has important consequences on the functioning of devices that rely on the physical exchange of information. Causal relations between events cannot be derived by simple comparison with an external, *a priori* given, parameter. For systems that are unlocalised in space, such as communicating processors, this means that the time-order relation of occurring events, even if it can be defined unambiguously at the local level of each processor, nevertheless requires a more complete representation to be defined over the whole system in a consistent way (Seitz 1980; Sutherland 1989). In the following, we shall analyse how the functioning of actual devices depends on the causal structure of physical space-time.

3. Logical devices

To discuss the intimate relation between time and computation, one must first recall some general principles that underlie the physical implementation of computing systems, and are applied in concrete machines realised with present technologies.

3.1. *Implementation of logical operations*

In CMOS (Complementary MOS) technology, logical gates are implemented using two electrical networks N_u and N_d, as represented in Figure 1. The x_i describe input channels,

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

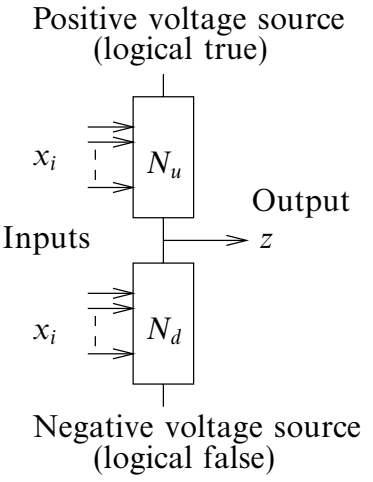

Fig. 1. A logic gate

and z the output channel. N_u and N_d are built with electrical switches, which are combined in series/parallel networks, thus allowing the implementation of the logics of propositions (Shannon 1938).

Each switch is implemented using a transistor. A function implemented by N will be said to be true, for a particular set of values of x_i, if and only if there is a path that connects the extreme connections of N. For instance, for the network represented by Figure 1:

— when N_u is *true*, output z is forced to value *true*.
— when N_d is *true*, output z is forced to value *false*.

The possibility that ($N_u = N_d = true$) for some values of x_i must be excluded, otherwise current could flow through both N_u and N_d, resulting in a short circuit between voltage sources. In the following, we shall always require that $(\neg N_u \vee \neg N_d)$ is verified for all configurations of variables x_i ($\neg$, $\vee$ and $\wedge$ define, as usual, negation, logical disjunction and logical conjunction).

Two cases must be considered:

1 N_u and N_d are always opposite, for all values of variables x_i, ($N_u = \neg N_d$). This case implements propositions of classical logic (Complementary MOS). The simple example of the *nand* gate is represented in Figure 2 (a bubble represents negation):

$$\begin{aligned} z = N_u &= \neg x_1 \vee \neg x_2 \\ N_d &= \neg N_u = x_1 \wedge x_2. \end{aligned} \tag{1}$$

2 N_u and N_d can be simultaneously *false*, that is, $(\neg N_u \wedge \neg N_d)$ can be *true*. In such configurations, the output z is not connected to any voltage source. Then, because

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

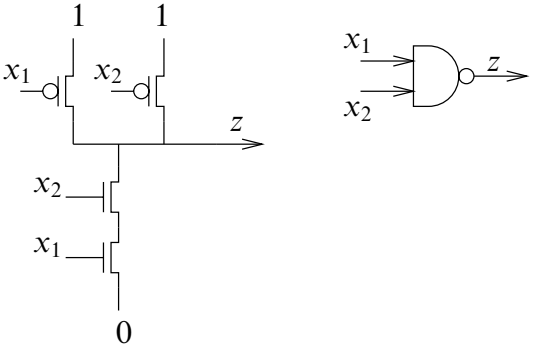

Fig. 2. Nand gate

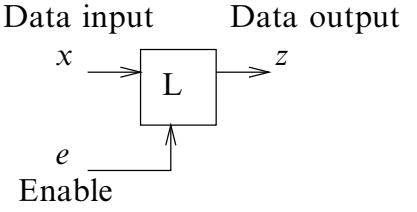

Fig. 3. Transparent latch

of electrical capacitance, z remembers its previous value. This allows us to implement memories, like the *latch* represented in Figure 3:

$$\begin{aligned} N_u &= x \wedge e \\ N_d &= \neg x \wedge e. \end{aligned} \tag{2}$$

In practice, in order to ensure the value is stored for long enough, and quite generally for all memories, one must ensure memory stability by using some feed-back, by means of a looped amplifier. This feed-back can be permanent (static) or recurrent (dynamic logic). One possibility of electrical feed-back is shown in Figure 4, where two looped amplifiers have been added on output z, one of them being weak, in the sense that it cannot create any serious short circuit when it conflicts with any of the two networks N_u and N_d.

Another way to implement a stable memory is to create static feed-back on a logical gate corresponding to the first case. Then, our *latch* can be realised using a looped multiplexer (*mux*), as shown in Figure 5:

$$\begin{aligned} z = N_u &= (x \wedge e) \vee (z \wedge \neg e) \\ N_d &= \neg N_u. \end{aligned} \tag{3}$$

This exhibits a very general difficulty that is characteristic of looped systems: variable z appears on both sides of its defining equation (3). This equation does not mean that

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

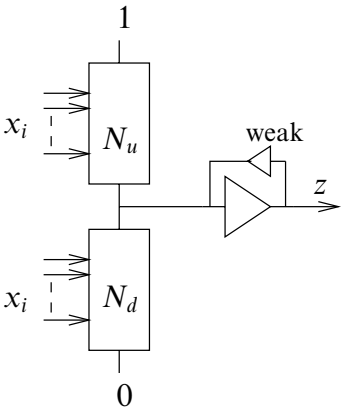

Fig. 4. Electrical feed-back

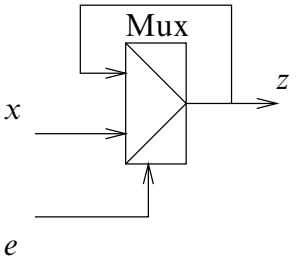

Fig. 5. Transparent latch with multiplexer

an equality must be realised, for instance with electric voltages, but that the following assignment must be realised:

$$z \leftarrow (x \wedge e) \vee (z \wedge \neg e) \tag{4}$$

$$z_a = (x \wedge e) \vee (z_b \wedge \neg e)\,. \tag{5}$$

In other words, we must distinguish two values of the variable z, which correspond to successive times: z_a (after) and z_b (before) are linked by equation (5). It is required that the two values z_a and z_b do not interfere, and that variable z changes from z_b to z_a. The assignment represented by equation (4) expresses a causality requirement that must be implemented in order to realise computations.

In the particular case of the *latch* just described, the operation can only cause a problem when e is falling. Indeed, in the other cases:

— when e is low, z is stored
— when e is high, z copies x
— when e rises, z begins to copy x.

Thus, the circuit operates correctly in these three cases. However, if e falls while x changes, z will hesitate between two values of x. The whole circuit may enter a metastable state,

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

which is invalid (the voltage will stay in metastable equilibrium at an intermediate level) and may last for an unbounded time. It may leave this state for any of the two possible values of z, and do this non-deterministically, which may not be acceptable for the type of computation envisaged. Let us note that the circuit of Figure 4 shows the same defects, since it involves a feed-back, although this may be less apparent when treated at the electrical level.

Although chosen here as an example, the *latch* shows properties that are encountered quite generally in looped devices. This brief discussion shows that the correct operation of a circuit cannot be analysed without taking its environment into account, in particular, the time ordering relations of input and output signals. This is entailed by the existence of loops and must be dealt with quite generally, since computing machines are naturally looped systems. In all cases, time constraints must be implemented in order to ensure the causality relations that are necessary for computation. These constraints will take very different forms, according to the type of implementation chosen, whether by means of synchronous or asynchronous systems. Before discussing separately these two classes of systems, we shall first recall one important property, the property of modularity, that they share since it is required for the implementation of complex computations.

3.2. *Modularity*

There are many various ways to organise electronic components into logic circuits, in order to realise machines performing computations. Usually, and quite generally, one defines complex circuits as hierarchies built with elementary circuits called primitives. This method, imposed by practical considerations, indeed hints at a logical necessity: one must be able to design and realise with the same rigour circuits of increasing complexity. More precisely, one must insure that circuits implementing logical functions of high complexity level behave as they should, and one must obtain this confidence in a rather short time. Because of the exponential increase in the number of configurations to check, this requirement implies that a direct physical test of the circuit's behaviour soon becomes impossible when the complexity of the logical function increases. This end can then only be attained with the help of modular implementations, by taking advantage both of their composite logical structure, and of the logical simplicity of the chosen primitives (Matherat and Jaekel 1996). Proofs relying on known properties of the composition of primitives may be developed, which allow one to deduce the correct functioning of a whole modular complex from that of its constituent primitives. Then, a test of the whole complex reduces to that of some of its constituents, which are logically simple. Although efficient, such a strategy may not reveal itself so straightforwardly. According to the type of physical implementation chosen for the primitives, problems may appear that prevent the systematic development of complex circuits operating correctly, and that do not occur when the choice of primitives is modified, or when peculiar constraints are put on their composition. Then, it follows that the logical and physical aspects of modular implementations must be analysed concurrently.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

4. Synchronous and asynchronous circuits

Time appears in computing systems very early, it is already there in the definition of the electronic circuits that implement logical functions. Most circuits that are known and used in practice are synchronous. Synchronous circuits may be defined as automata whose transitions between successive states are triggered by pulses delivered by a global clock. Asynchronous circuits provide an alternative class of circuits. In this section, we introduce the strategies followed by these two main classes of circuits for making the implementation of computation effective, and, in particular, for dealing with problems of computation interferences.

4.1. *Synchronous circuits*

VLSI circuits that are produced nowadays are highly concurrent devices (a microprocessor can contain up to 10^7 transistors, that is, 10^6 logical gates), and yet most of them can be modelled as a non-concurrent device and can be considered as a single finite automaton. This property comes from their synchronous character, which means that all operations on internal memories are simultaneously activated by a single pulse of a global clock shared by the whole circuit.

Synchronous implementations use a global clock to avoid the stability problem that we discussed in the previous section. More precisely, with the same notation, all *latches* are systematically operated in such a way as to ensure that logical variables x are stable when variables e are falling. There exist many different types of memories, but all present the same problem, reflecting the time character of logical assignment. For simplicity, we shall only discuss the case of the *latch* considered earlier, which can be regarded as a generic example. A synchronous device using two items of this *latch* for each register bit (a master-slave *flip-flop*) is sketched in Figure 6. The enabling signals e_1 and e_2 are mutually excluded in time, and are derived systematically from a common clock. The output is fed back under the form of input variables Q_i into a combinatorial operator. If the clock period is larger than the feedback time, the variables Q'_i are always stable when e_1 is falling and the *latches* act as required, that is, they make the iteration of the combinatorial function effective.

The global state Q is encoded by the state of all memory bits, and can only change at the arrival of a clock pulse. In terms of their specification and design, synchronous circuits may be considered as modular composites, where primitives, and other modules as well, are finite automata of the type described by Figure 6. The register encodes the state of the circuit, while the combinatorial operator represents the implementation of its transition function. All registers are activated by a single clock pulse. By connecting several automata of this type, one obtains another automaton of the same type, but with a larger memory and a more complex combinatorial function. In such a representation, and from a logical point of view, neither time nor space are involved. One only needs to consider the successive logical steps associated with successive clock periods. The logical time of a computation reduces to a mere integer, which one only relates to physical time by multiplying it by the mean clock period. In other words, time is discretised.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

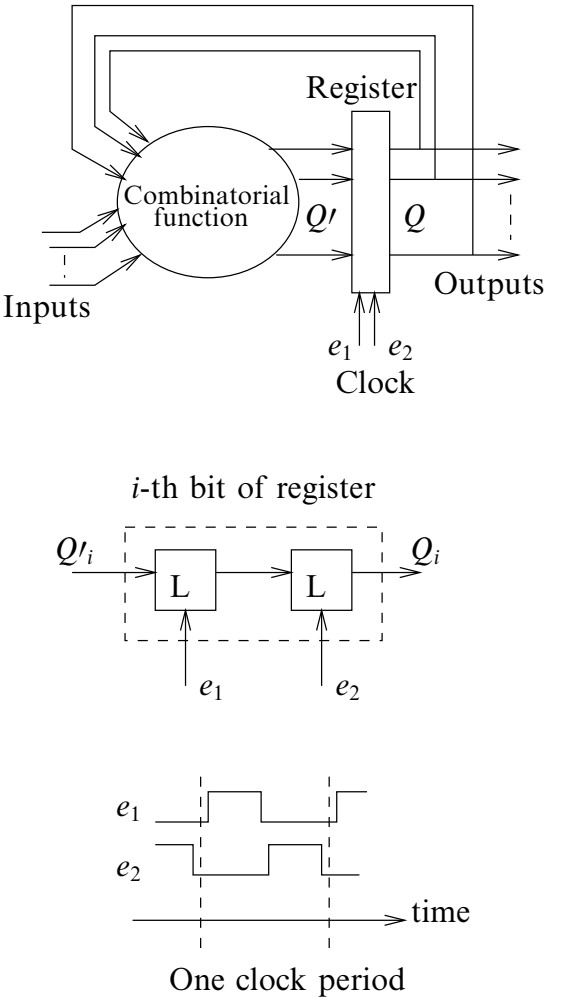

Fig. 6. Synchronous circuit

The only physical constraint one must impose is that the clock frequency must be smaller than the limit value necessary for all internal propagations to be performed in less time than the clock period. Space plays no role in the logical function. The whole circuit may be considered as local, that is, propagation times need only be taken into account when circuits are connected over large distances, that is, when propagation times are large compared to the clock period, as is the case when computers are connected. Implementation on a silicon chip must take into account and control all propagation times within a circuit, in order to ensure that all inputs become stable before the end of each clock period. The clock signal must be implemented so that it arrives simultaneously at all *latches*, that is, with negligible delays when compared to the clock period. Clearly, such properties can only be checked once the whole circuit has been specified. Such a

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

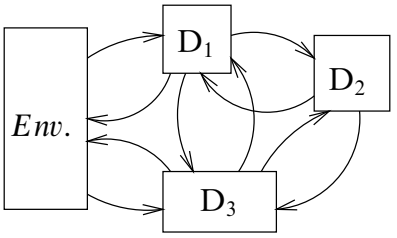

Fig. 7. Asynchronous circuit

circuit cannot be implemented incrementally, that is, by implementing a part without any knowledge of its connections with other parts or of the clock frequency of the whole circuit. In other words, all parts must be local at every scale, from primitives to the whole circuit.

The synchronous approach is, however, being questioned in present VLSI designs. This arises mainly because of difficulties encountered when distributing simultaneously the same clock signal to millions of *latches*, over several cm^2, and at a frequency of the order of a Gigahertz. Clock distribution uses a significant proportion of the chip surface and contributes an important part to the overall dissipation.

4.2. *Asynchronous circuits*

Asynchronous circuits may be defined in opposition to synchronous circuits, by the requirement that they do not use a global clock. However, rather than being complementary, as a class they include synchronous circuits.

Models usually represent asynchronous circuits as general devices that are distributed and communicate along connecting channels, as shown in Figure 7. The activity of such circuits is not ruled by the pulses of a global clock, but proceeds through communications distributed between many concurrent parts. These devices can be simple logical gates (a few transistors) or, at the opposite extreme, complex processors. Communications can be realised through a single wire or through a complex network. Clearly, concurrency can no longer be ignored. Indeed, one can no longer define a logical state that would be associated with the global circuit at a definite time. This is because each device can change its state following a communication, without being synchronised with most other devices. The notion of a computing step itself must be revised, as it relies on a total ordering of all logical events.

Problems of computation interference may then arise at two different levels. At the lowest level, the correct functioning of a single component may be endangered by computing interferences within the component itself, due to internal loops and instabilities of internal variables. At the highest level, the composition of asynchronous circuits may induce computation interferences due to exchanges between one module and its environment, the latter sending signals that conflict with the module operation.

We will discuss the second case only, and assume that primitives may be defined that are free from internal computing interferences (see, for instance, Molnar *et al.* (1985)

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

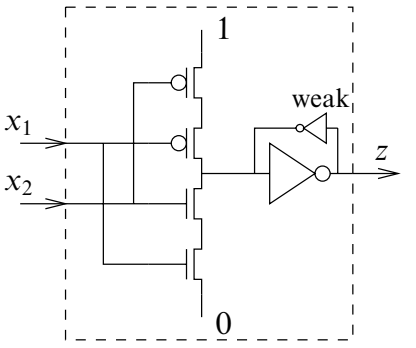

Fig. 8. Simplified C-element

and Furber (1995)). First, we will briefly describe those that are most frequently used in asynchronous circuits. In some examples, logical functions are defined in a way that does not distinguish between rising and falling edges. These undistinguished transitions are called events, and the logical function operates on these events. But the events are still transitions between different levels (or Boolean variables), so each primitive can be considered in both ways: either as an operation on Boolean variables or as (another) operation on events.

A very frequently used primitive is the *join* element, or Muller's *C-element*. It has two inputs x_1 and x_2 and one output z, and its logical function can be described as follows:

— if $x_1 = x_2$, then $z = x_1 = x_2$
— if inputs become different from one another, z keeps its previous value.

Then, ouput z only changes after both inputs x_1 and x_2 have changed. This allows a rendezvous to be realised between levels (wait until two inputs acquire the same value) or between events (wait until two inputs have received the same number of rising or falling edges, after proper initialisation).

The primitive *C-element* can be realised using electrical feed-back analogous to that of Figure 4. A common realisation following these lines is represented in Figure 8.

Another primitive is the *toggle*, represented in Figure 9, which has one input x and two ouputs z_1 and z_2. Successive events on the input are alternately sent to outputs z_1 and z_2. The first event after initialisation is sent to the marked output z_1.

The *or* operation between events, also called *merge*, can be realised using a classical *exclusive or* gate (*xor* between levels). The *sequencer*, represented in Figure 9, possesses three inputs x_1, x_2 and x_3 and two outputs z_1 and z_2. Its role is to grant a given resource to one of two different processes, which can make requests on inputs x_1 and x_2. When an event is received on x_3, a granting event is produced either on z_1 or z_2 according to an existing request on x_1 or x_2, respectively. When two requests are present, the *sequencer* arbitrates between the two, and thus introduces a component of indeterminism. The *sequencer* may take an unbounded time to arbitrate, but it is required to realise the mutual exclusion of the two grant signals.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

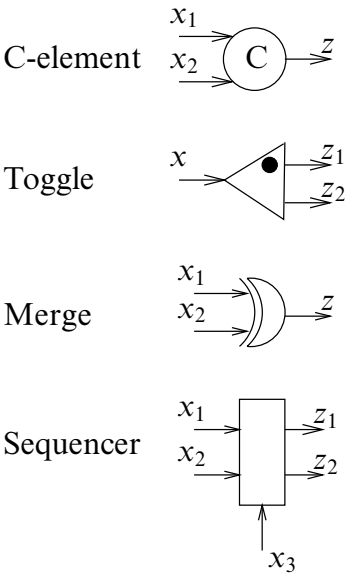

Fig. 9. Some asynchronous primitives

5. Composition of circuits and time ordering

In this section, we discuss the composition of asynchronous circuits, and some solutions that have been brought to the problem of computing interferences.

Compositions of asynchronous circuits correspond to distributed systems, where different parts communicate in a way that is not regulated by a global clock. Then various and arbitrary time delays affect successive transitions at the input of a module. Inputs may then conflict with the correct operation of the module itself. The approaches followed in circuit design to deal with computing interferences fall into two main classes. One practical approach to timing problems consists of working directly on the physical implementation, by keeping track of all the delays occurring in the logical circuit, together with all the constraints that must be satisfied by these delays in order to make the whole circuit operate correctly. Then, programs are developed to find and optimise solutions systematically (Chakraborty *et al.* 1988; Stevens *et al.* 1999). Although practically very efficient, this strategy rapidly attains such a complexity that it becomes very difficult to distinguish fundamental issues from practical choices. In the other class of approaches, one attempts to separate as much as possible the logical issues related to timing from their physical manifestations, which arise mainly from the values of time delays. This has led to different studies, focussing either on the determination of the best choice of logical primitives, satisfying criteria such as speed-insensitivity or delay-insensitivity (Martin 1990a), or on a more restrictive definition of modular composition, such as delay-insensitive compositions (Udding 1986; Ebergen 1991). In the following, we shall only briefly discuss approaches related to delay-insensitivity, and focus on the fundamental relation they tend to exhibit

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

between the occurence of computing interferences and the causal structure of physical space-time.

In order to make the analysis easier to follow, we shall introduce a graphical representation of the communications occurring between modules of a composition (see Figure 11). These graphs are analogous to those that can be used in relativistic physics to represent the space-time evolution of localised physical systems, together with the light signals they exchange. As discussed in a previous section, an essential feature is the absence of an *a priori* given global and common time. Only a local time ordering can be made between the successive events occurring on each module, reflecting the causal relations that can be made locally. Although time is represented as the vertical axis, this only indicates the direction for increasing time on each module. Different modules are displayed on the horizontal axis, which roughly corresponds to space. Each module is then represented by a vertical line, indicating the causal succession of the local events occurring at its inputs or outputs. Communications are then represented by inclined arrows leaving a module (output) and arriving at another module (input). Although they may vary, the slopes of these arrows must always be greater than a strictly positive lower bound, which corresponds to the velocity of light. Varying slopes indicate that varying speed and delays affect communications between modules.

In the following, we shall call each arrow corresponding to a communication an 'event', and the intersection of this event with the time evolution of a module a 'point' (thus following the notation introduced by Russell in his discussion of the causal structure of relativistic space-time (Russell 1927)). The logical specification of each module is translated into causal relations between the points that represent the occurrence of events on the module. These local constraints may be given a precise expression using a formal language that is well suited to representing time-ordered event structures (Martin 1990a; Ebergen 1991; Winskel 1989). As propagation delays play an essential part, ordering constraints will be most conveniently visualised using graphical representations, which allow the analysis of global causal relations within distributed systems.

5.1. *Delay-insensitivity*

In order to discuss the role of delays in computation interferences, we will first analyse the example of the *Q-element* (Martin 1990a; Martin 1990b), which is represented in Figure 10. The formal expression describing the logical function of the *Q-element* can be written in a language derived from CSP (Communicating Sequential Processes) (Hoare 1978):

$$* [[x_i]; y_o \uparrow; [y_i]; u \uparrow; [u]; y_o \downarrow; [\neg y_i]; x_o \uparrow; [\neg x_i]; u \downarrow; [\neg u]; x_o \downarrow]. \quad (6)$$

Each variable between brackets, which precedes a transition, represents a logical variable that must be true before the circuit can execute the transition that follows (; denotes time succession, and * arbitrary repetition of the expresssion in brackets). Thus, the circuit waits for x_i to be true, then emits a rising edge on output y_o, and so on.

This logical function can be implemented as a composition of a *C-element* with two *and* gates, as represented in Figure 10. Output u of the *C-element* is followed by a fork,

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

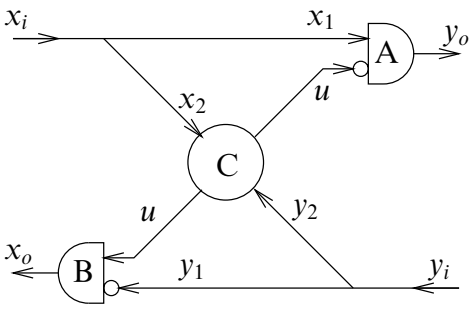

Fig. 10. Q-element

which relates u to one input of each of the *and* gates. Two other forks also dispatch the event produced by the environment x_i (respectively, y_i) on two inputs denoted by x_1 and x_2 (respectively, y_1 and y_2).

The logical operation of the circuit may be represented using a space-time graph, as in Figure 11. The left and right parts of the circuit environment are represented as X and Y, respectively. The series of points corresponding to the definition of the logical function of each module can be followed on each vertical line. Situations that correspond to rendezvous, that is, intervals where a primitive is waiting for the arrival of two events in any order, have been represented by a thick line. This is systematically the case for the *C-element*, but also for the *and* gates, when they are waiting for their two inputs to be true. Pairs of points that cannot occur in reverse order without ruining the computation, have been shown by dashed lines. The two cases involve the internal variable u and one event, $y_1 \uparrow$ or $x_1 \downarrow$ produced by the environment (Y or X). Event $y_1 \uparrow$ must reach the *and* gate B before event $u \uparrow$, recalling that the latter has been produced by the arrival of event $y_2 \uparrow$ on the *C-element*. Then, the fork that dispatches both events $y_1 \uparrow$ and $y_2 \uparrow$ plays a crucial role in determining the order of points on the *and* gate B.

A few remarks are in order. Concurrent computing is well illustrated by Figure 11. Different computations proceed along paths involving vertical and propagation lines, each representing a causally ordered series of operations. Causal order only makes sense either within each vertical line, where it is associated with the logical function of the module, or within propagating lines, where it connects the output of one module to the input of another module. But no *a priori* total order exists between all points of the graph. This is illustrated by the independence of computation on the order of some pairs of points. For instance, two events belonging to different branches of the fork on variable u at the output of the *C-element* may have arbitrary relative order.

Imposing a total ordering would amount to implementing a global time, by means of clock distribution for instance, which would allow one to draw horizontal lines on the graph of Figure 11. But such a condition is too restrictive, as computation only relies on causal relations imposed by vertical and propagation lines. The remaining freedom in the ordering of events, such as the one related to the fork at the output of the *C-element*,

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

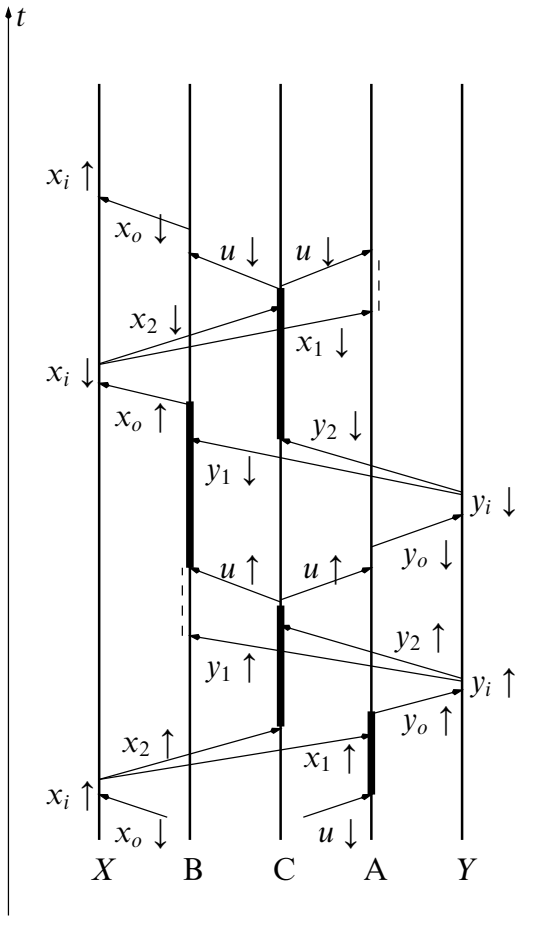

Fig. 11. Space-time graph of the Q-element

is necessary for optimising the circuit performance. For a definite implementation, event ordering will depend on the relative spatial position of modules, so the remaining freedom may be used to find an optimal arrangement of modules on the chip.

The property of delay-insensitivity (Molnar *et al.* 1985) is easily seen on the graph. It corresponds to the independence of causal ordering of computation on delays occurring in responses of modules or in propagations of signals, that is, on the vertical or horizontal displacements of the modules. Such a property is made possible by using primitives that wait for the arrival of events at their input before producing other events at their output. But this condition appears to be insufficient. In this respect, it is instructive to compare the two kinds of forks used by the previous composition implementing the *Q-element*. No constraint affects the events produced by the fork at the output of the *C-element* (thick lines in Figure 11). However, forks dispatching the events produced by the environment must be implemented in such a way as to respect the causal order of the events that they generate and that finally arrive at the same *and* gate (dashed lines in Figure 11). Such

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

forks, which are called *isochronic forks* (Martin 1990a; Martin 1990b), must be isolated and given a special treatment at the implementation level, in order to satisfy the delay constraints that are necessary for preserving causal ordering.

The property of delay-insensitivity (DI) has been introduced and much developed as a simple condition one can impose on primitives and logical circuits, with the aim of designing asynchronous circuits of arbitrary complexity systematically, without having to take time scales into account. One approach consists of defining DI circuits as compositions of stable primitives devoid of internal loops (except for electrical loops used as memory) (Martin 1986). A primitive is defined to be stable, by requiring that an input that changes the output cannot change before the output has been established. It can then be shown that only compositions of *C-elements* can be DI according to this definition.

But, it can also be shown that compositions using *C-elements* (and generalised *C-elements* with more inputs) exclusively, strongly limit the type of allowed computations, and exclude most circuits of interest (Martin 1990b). The *isochronic fork* may then be advocated as a weakest compromise to delay insensitivity. Adding the *isochronic fork* and using this extended class of elements, which are called quasi delay-insensitive (QDI), complex and efficient asynchronous circuits have been realised (Martin *et al.* 1997). However, as illustrated by the example of the *Q-element*, *isochronic forks* need to be identified at the logical level and their implementation must be given a special treatment, which may become intricate for very complex circuits.

5.2. *Delay insensitive composition*

Another approach to avoiding computation interferences (Molnar *et al.* 1985; Udding 1986; Ebergen 1991) consists of defining a less restrictive set of DI primitives, together with a notion of DI composition of these primitives. Circuits are represented in a formal language, called trace theory, similar to the one used in equation (6), with further syntax rules on logical operations. Computing interferences are avoided by imposing structural constraints under the form of simple rules. Let us first recall some definitions and properties of trace structures (Ebergen 1991; Mazurkiewicz 1989).

Definition 1. Trace structures are defined as triples $R =< \mathbf{i}R, \mathbf{o}R, \mathbf{t}R >$, where $\mathbf{i}R$ and $\mathbf{o}R$ are finite sets of symbols (the input and output alphabets, respectively), and $\mathbf{t}R$ is the set of traces, which is a subset of $(\mathbf{i}R \cup \mathbf{o}R)^*$, the set of all finite-length sequences of symbols taken in the union set $\mathbf{i}R \cup \mathbf{o}R$.

Trace structures are traditionally denoted by capital letters, while lower case letters a, b, c denote symbols and s, t traces. The following abbreviations are also frequently used:

$$\begin{aligned} a? &\equiv < \{a\}, \varnothing, \{a\} > \\ b! &\equiv < \varnothing, \{b\}, \{b\} >. \end{aligned} \tag{7}$$

Operations of concatenation, union, repetition, prefix-closure, projection and weaving are defined on trace structures.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

Definition 2.

$$
\begin{aligned}
&R;S \equiv < \mathbf{i}R \cup \mathbf{i}S, \mathbf{o}R \cup \mathbf{o}S, (\mathbf{t}R)(\mathbf{t}S) > \\
&R|S \equiv < \mathbf{i}R \cup \mathbf{i}S, \mathbf{o}R \cup \mathbf{o}S, \mathbf{t}R \cup \mathbf{t}S > \\
&*[R] \equiv < \mathbf{i}R, \mathbf{o}R, (\mathbf{t}R)^* > \\
&\mathbf{pref}\, R \equiv < \mathbf{i}R, \mathbf{o}R, \{t_0 | \exists t_1 : t_0 t_1 \in \mathbf{t}R\} > \\
&R \downarrow A \equiv < \mathbf{i}R \cap A, \mathbf{o}R \cap A, \{t \downarrow A | t \in \mathbf{t}R\} > \\
&R||S \equiv < \mathbf{i}R \cup \mathbf{i}S, \mathbf{o}R \cup \mathbf{o}S, \\
&\qquad \{t \in (\mathbf{a}R \cup \mathbf{a}S)^* | t \downarrow \mathbf{a}R \in \mathbf{t}R \wedge t \downarrow \mathbf{a}S \in \mathbf{t}S\} >
\end{aligned}
\tag{8}
$$

where, for convenience, the notation $\mathbf{a}R \equiv \mathbf{i}R \cup \mathbf{o}R$ has been introduced for the total alphabet of R, where $t \downarrow A$ denotes the projection of trace t on alphabet A and $(\mathbf{t}R)^*$ is the set of all finite-length concatenations of traces in $\mathbf{t}R$ (symbols $\exists$, $\forall$, $\in$, $\cap$ and $\cup$ denote, as usual, existence, universality, set membership, set intersection and set union). The **pref** operator constructs prefix-closed structures, while the projection operator hides internal symbols; finally, the weave operator expresses instantaneous synchronisation. A circuit is specified by a prefix-closed, non-empty, trace structure R with $\mathbf{i}R \cap \mathbf{o}R = \varnothing$. The trace structure representing the environment of a circuit with trace structure R is the reflection of the latter, and may also be given a compact notation:

$$
\bar{R} = < \mathbf{o}R, \mathbf{i}R, \mathbf{t}R >. \tag{9}
$$

A trace structure R may be implemented physically by letting each symbol a in the alphabets $\mathbf{i}R$ and $\mathbf{o}R$ correspond to a channel, and each occurence of this symbol in a trace of $\mathbf{t}R$ correspond to an event, that is, a high or low transition, on the corresponding channel. Symbols in $\mathbf{i}R$ or $\mathbf{o}R$ describe communication actions that are produced by the environment or (exclusive or) the circuit, respectively.

In order that we can ignore transmission delays while avoiding transmission and computing interferences, the following rules may be imposed (Udding 1986):

$$
\begin{aligned}
&\mathbf{R_0} \quad \forall s \in \mathbf{t}R, \quad a \in \mathbf{a}R \qquad saa \notin \mathbf{t}R \\
&\mathbf{R_1} \quad \forall s,t \in \mathbf{t}R, \quad (a,b \in \mathbf{i}R) \vee (a,b \in \mathbf{o}R) \\
&\qquad\qquad sabt \in \mathbf{t}R \quad \Leftrightarrow \quad sbat \in \mathbf{t}R \\
&\mathbf{R_2} \quad \forall s,t \in \mathbf{t}R, \quad (a \in \mathbf{i}R \wedge b \in \mathbf{o}R) \vee (a \in \mathbf{o}R \wedge b \in \mathbf{i}R) \\
&\quad (sab \in \mathbf{t}R \wedge sba \in \mathbf{t}R) \quad \Rightarrow \quad (sabt \in \mathbf{t}R \Leftrightarrow sbat \in \mathbf{t}R).
\end{aligned}
\tag{10}
$$

Rule $\mathbf{R_0}$ excludes two consecutive transitions on the same wire, and hence the transmission interferences that may result. Rule $\mathbf{R_1}$ expresses independence of computation on the order of signals travelling in the same direction, as this order may depend on any delays suffered. The *C-element* is easily seen to satisfy this rule. However, the *and* gate only complies with the rule when it is waiting for a rising edge on its two inputs, and in all other cases does not. Thus the *and* gate, and the *or* gate, are excluded by this rule, although the *toggle*, the *merge* and the *sequencer* are compatible. Rule $\mathbf{R_2}$ expresses the same property for signals

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

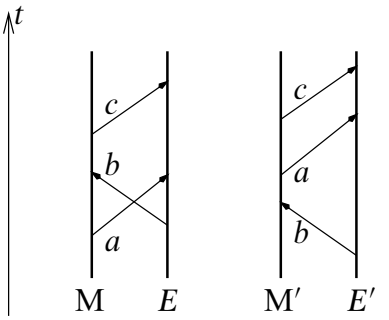

Fig. 12. Space-time graph for rule R_2'

travelling in opposite directions, when their order does not change the result locally. Note that due to the necessarily symmetric treatment of a circuit and its environment, all rules are symmetric in the exchange of input and output symbols.

One must exclude the possibility that a symbol of one type might disable a symbol of another type (symbol a is said to disable symbol b in trace structure R if there is a trace s with $sa \in \mathbf{t}R \wedge sb \in \mathbf{t}R \wedge sab \notin \mathbf{t}R$). This exclusion is necessary to prevent an admissible input symbol being disabled by an output signal, depending on the delay the former has suffered on its way to the circuit (by symmetry the same property must also hold for the environment and output signals). Depending on the level of exclusion, this property leads us to define three classes, with rules $\mathbf{R}_3'$, $\mathbf{R}_3''$ and $\mathbf{R}_3'''$:

$$\begin{aligned} \mathbf{R}_3' \quad & \forall s, \quad a \neq b \in \mathbf{a}R \\ & (sa \in \mathbf{t}R \wedge sb \in \mathbf{t}R) \quad \Rightarrow \quad sab \in \mathbf{t}R \end{aligned} \tag{11}$$

$$\begin{aligned} \mathbf{R}_3'' \quad & \forall s, \quad a \neq b \in \mathbf{a}R \quad a \notin \mathbf{i}R \vee b \notin \mathbf{i}R \\ & (sa \in \mathbf{t}R \wedge sb \in \mathbf{t}R) \quad \Rightarrow \quad sab \in \mathbf{t}R \end{aligned} \tag{12}$$

$$\begin{aligned} \mathbf{R}_3''' \quad & \forall s, \quad (a \in \mathbf{i}R \wedge b \in \mathbf{o}R) \vee (a \in \mathbf{o}R \wedge b \in \mathbf{i}R) \\ & (sa \in \mathbf{t}R \wedge sb \in \mathbf{t}R) \quad \Rightarrow \quad sab \in \mathbf{t}R. \end{aligned} \tag{13}$$

These rules allow successively for greater decision possibilities. Rule $\mathbf{R}_3'$ does not permit data transmission and is called a synchronisation class (an example is provided by the *C-element*). Rule $\mathbf{R}_3''$ allows for two inputs to disable each other and is called a data communication class. With rule $\mathbf{R}_3'''$, a circuit may choose between two output symbols and belongs to the arbitration class.

Finally, rule $\mathbf{R_2}$ appears to be too restrictive in specific examples (Udding 1986). An alternative and more generally efficient rule is provided by:

$$\begin{aligned} \mathbf{R}_2' \quad & \forall s, t \in \mathbf{t}R, \quad (a, c \in \mathbf{i}R \wedge b \in \mathbf{o}R) \vee (a, c \in \mathbf{o}R \wedge b \in \mathbf{i}R) \\ & (sabtc \in \mathbf{t}R \wedge sbat \in \mathbf{t}R) \quad \Rightarrow \quad sbatc \in \mathbf{t}R. \end{aligned} \tag{14}$$

This rule, which is conveniently expressed on a space-time graph, as shown in Figure 12, concerns three events a, b, c connecting one module M and its environment E. It stipulates

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

Table 1. *DI primitive components.*

CIRCUIT	Specification
WIRE	**pref** $*$ $[a?;b!]$
IWIRE	**pref** $*$ $[b!;a?]$
FORK	**pref** $*$ $[a?;b!\|\|c!]$
C-ELEMENT	**pref** $*$ $[a?\|\|b?;c!]$
TOGGLE	**pref** $*$ $[a?;b!;a?;c!]$
MERGE	**pref** $*$ $[(a?\|b?);c!]$
SEQUENCER	**pref** $*$ $[a?;p!]$ $\|\|$**pref** $*$ $[b?;q!]$ $\|\|$**pref** $*$ $[n?;(p!\|q!)]$

that, if two time orders are allowed for the occurences of two events of different types (that is, one input and one output) a and b, if the event c, of the same type as a, is a consequence of the order 'a then b', it should also be a consequence of the other order 'b then a'. This rule imposes the condition that if, because of propagation time delays, an order on events is seen differently by a module and its environment, this order should have no consequence on the logical behaviour of the module. As illustrated by Figure 12, this rule only affects the case on the left part of the figure, that is, only the case when propagation can change the order of events.

The set of DI components is given by trace structures, defined according to Definitions 1 and 2, that satisfy the weakest form of the rules, that is, R_0, R_1, R_2' and R_3''' (Udding 1986).

In this way, a set of DI primitive components for asynchronous circuits can be obtained with the following list of specifications in terms of trace structures (see Table 1).

The *wire* corresponds to a component that waits for an event to occur on its input, then sends an event on its output, and repeats this sequence indefinitely. The inverted wire (*iwire*) behaves similarly, but begins by sending an event on its output. The *fork* duplicates one input. As can be seen from definitions (8), weaving not only consists of putting in parallel, but also in synchronising common output symbols. In the particular case of two *wires* with a common output, weaving leads to the *C-element*. The other components correspond to the primitive circuits that have been previously introduced (see Figure 9).

The objective is to realise circuits corresponding to given complex specifications by combining simple DI primitive circuits. This aim may be attained by making use of operations such as decomposition and substitution, together with two theorems setting the conditions for performing these operations.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

Definition 3. A component R_0 is said to be decomposed into components $R_i, 1 \leqslant i < n$ (for $n > 1$) if the following conditions are satisfied. Letting $S_0 = \bar{R}_0, \quad S_i = R_i \quad \text{for} \quad 1 \leqslant i < n, \quad \text{and} \quad T = ||_{0 \leqslant i < n}(S_i)$

$$\begin{aligned}
&\text{(i)} \quad \cup_{0 \leqslant i < n}(\mathbf{o}S_i) = \cup_{0 \leqslant i < n}(\mathbf{i}S_i) \\
&\text{(ii)} \quad \mathbf{o}S_i \cap \mathbf{o}S_j = \varnothing, \quad \text{for} \quad 0 \leqslant i, j < n \quad \text{and} \quad i \neq j \\
&\text{(iii)} \quad \forall t, x, i \quad (0 \leqslant i < n) \\
&\qquad t \in \mathbf{t}T \wedge x \in \mathbf{o}S_i \wedge tx \downarrow \mathbf{a}S_i \in \mathbf{t}S_i \Rightarrow tx \in \mathbf{t}T \\
&\text{(iv)} \quad \mathbf{t}T \downarrow \mathbf{a}S_0 = \mathbf{t}S_0
\end{aligned} \tag{15}$$

The conditions in (15) describe:

(i) a closed network (each input is connected to an output, and conversely),
(ii) the absence of output interferences (two outputs cannot be connected),
(iii) the absence of computing interferences (any event produced by a component is compatible with the behaviour of the component that receives it)
(iv) the correct behaviour at the circuit boundary (network behaves as prescribed).

Decomposition will be denoted by $R_0 \to (R_i)_{1 \leqslant i < n}$.

Let us now state two useful theorems (proofs may be obtained in Ebergen (1987)).

Theorem 1 (Substitution Theorem). For components R_0, R_1, R_2, R_3 and S

$$\begin{aligned}
&R_0 \to (R_1, S) \quad \wedge \quad S \to (R_2, R_3) \\
&\Rightarrow \qquad R_0 \to (R_1, R_2, R_3)
\end{aligned}$$

holds if

$$(\mathbf{a}R_0 \cup \mathbf{a}R_1) \cap (\mathbf{a}R_2 \cup \mathbf{a}R_3) = \mathbf{a}S. \tag{16}$$

The latter condition stipulates that internal symbols of S, that is, symbols in $(\mathbf{a}R_2 \cup \mathbf{a}R_3) \backslash S$, where $\backslash$ means set deletion, should not appear in $(\mathbf{a}R_0 \cup \mathbf{a}R_1)$. It can be realised by an appropriate renaming of the internal symbols of S.

Theorem 2 (Separation Theorem). For components R_i and S_i $(0 \leqslant i < n)$,

$$\begin{aligned}
&R_0 \to (R_i)_{1 \leqslant i < n} \quad \wedge \quad S_0 \to (S_i)_{1 \leqslant i < n} \\
&\Rightarrow \qquad R_0 || S_0 \to (R_i || S_i)_{1 \leqslant i < n}
\end{aligned}$$

holds if

$$\begin{aligned}
&(\cup_{1 \leqslant i < n}(\mathbf{a}R_i) \backslash \mathbf{a}R_0) \cap (\cup_{1 \leqslant i < n}(\mathbf{a}S_i) \backslash \mathbf{a}S_0) = \varnothing \\
&\quad \text{and,} \qquad \text{for} \quad 1 \leqslant i \neq j < n
\end{aligned} \tag{17}$$

$$\begin{aligned}
&(\mathbf{o}R_i \cup \mathbf{o}S_i) \cap (\mathbf{o}R_j \cup \mathbf{o}S_j) = \varnothing \\
&(\mathbf{o}R_i \cup \mathbf{o}S_i) \cap (\mathbf{o}\bar{R}_0 \cup \mathbf{o}\bar{S}_0) = \varnothing.
\end{aligned} \tag{18}$$

Condition (17) stipulates that the internal symbols of the decompositions of R_0 and S_0 are disjoint (this condition may be satisfied by renaming some of these symbols), and

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

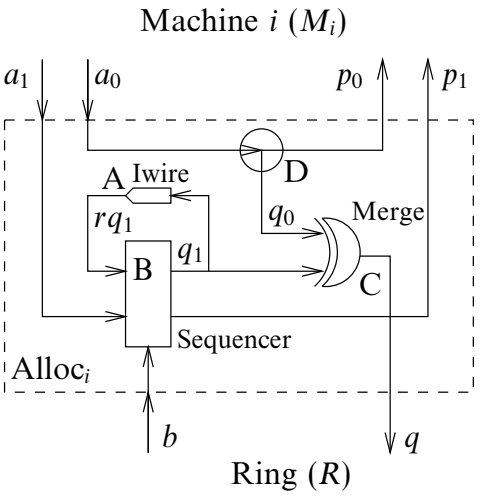

Fig. 13. Token-ring interface

conditions (18) stipulate that the outputs of any two components $R_i||S_i$ and $R_j||S_j$ are also disjoint when the components are different (these conditions may also be satisfied by reordering the components).

With the help of these two theorems, the previously defined DI primitives may be combined to give modular compositions that are delay insensitive, hence circuits where computing interferences cannot be introduced by delay modifications only. We briefly describe an example of a circuit that can be obtained with such a composition of DI primitives, the *token-ring* interface (Ebergen 1991). The *token-ring* interface is a device allowing us to connect several machines, which must share a common resource (like a memory, or a bus). One item Alloc$_i$ of this device will be associated with each machine M_i, all items being identical and realising the same function, as shown by Figure 13. Item Alloc$_i$ of this device is connected to two environments, the machine M_i on top of the figure, and, at bottom, the ring R where a token circulates. The arrival of the token at Alloc$_i$ corresponds to an event on b, its departure to an event on q. The machine M_i can make a request in the form of an event on a_1. Alloc$_i$ grants the resource to the machine M_i by an event on p_1. The machine M_i signals the end of its use of the resource by an event on a_0, which is acknowledged by Alloc$_i$ in the form of an event on p_0.

Initially, the *token-ring* interface is specified by the following trace structure:

$$\begin{aligned} &\mathbf{pref} * [a1?; p1!; a0?; p0!] \\ ||&\mathbf{pref} * [b?; (q!|p1!; a0?; q!)] \end{aligned} \tag{19}$$

This specification results from weaving two trace structures that describe the communications of the token-ring interface with the machine M_i and with the ring R, respectively. The two trace stuctures interact through their common dependence on two events $p1$ and

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

$a0$. Each trace structure may be decomposed into primitive elements. The substitution and separation theorems may be applied, leading finally to a possible decomposition, as shown graphically by Figure 13:

$$\begin{aligned} \rightarrow \quad (\quad & \mathbf{pref} * [a1?; p1!] \\ & \|\mathbf{pref} * [rq1?; q1!] \\ & \|\mathbf{pref} * [b?; (q1!|p1!)], \\ & \\ & \mathbf{pref} * [rq1!; q1?], \\ & \mathbf{pref} * [a0?; p0!], \\ & \mathbf{pref} * [a0?; q0!], \\ \mathbf{pref} * & [(q1?|q0?); q!)] \quad) \end{aligned} \tag{20}$$

The first component is a *sequencer* (see Table 1), which is necessary for synchronising the output $p1$ shared by the two trace structures defining the *token-ring*. The *sequencer* also arbitrates between corresponding inputs. Other components describe an *iwire*, two *wires* and a *merge*. Although they do not appear explicitly in decomposition (20), two forks appear in Figure 13, as a consequence of double occurences of $a0?$ and $q1?$ in (20).

The DI property of this implementation can be seen graphically on a space-time graph, as in Figure 14. Two cases are shown in Figure 14. In the first case, the request a_1 made by the machine i is not granted, the token being sent back to the ring. When the token arrives a second time, the resource is granted to the machine M_i, which was waiting. This illustrates the non-deterministic behaviour of the module Alloc, which depends on arbitration performed by the *sequencer* B. The figure also shows that the two forks, that on q_1 (output of B) and that defined by D, cannot create computation interferences, so that no particular constraints are necessary. This results from the function of the *sequencer* B, which is not perturbed whatever the order of the events on its inputs. The *sequencer* B waits for an event on b to make a decision, and then arbitrates between the different requests it has received.

As shown by the example of the *token-ring* interface, DI primitives and DI decomposition may be used to generate modular compositions that are delay insensitive, and, as shown with the help of space-time graphs, remain free of computing interferences. Delay-insensitivity appears as a simple criterion for escaping problems raised by computing interferences in a purely logically way, that is, without recourse to a detailed analysis of the physical implementation of a circuit. The DI criterion allows one to treat asynchronous circuits efficiently, as in the case of synchronous circuits, by allowing us to represent them formally (in terms of trace structures). Although revealing a genuinely different underlying structure, the causal constraints on asynchronous circuits, as exhibited by space-time graphs, can, nonetheless, be embedded in a simple set of formal rules that limit the definition and composition of DI circuits. In general, these rules allow DI circuits to be decomposed into a number of DI primitive components that increases linearly with the length of the circuit specification (Ebergen 1987).

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

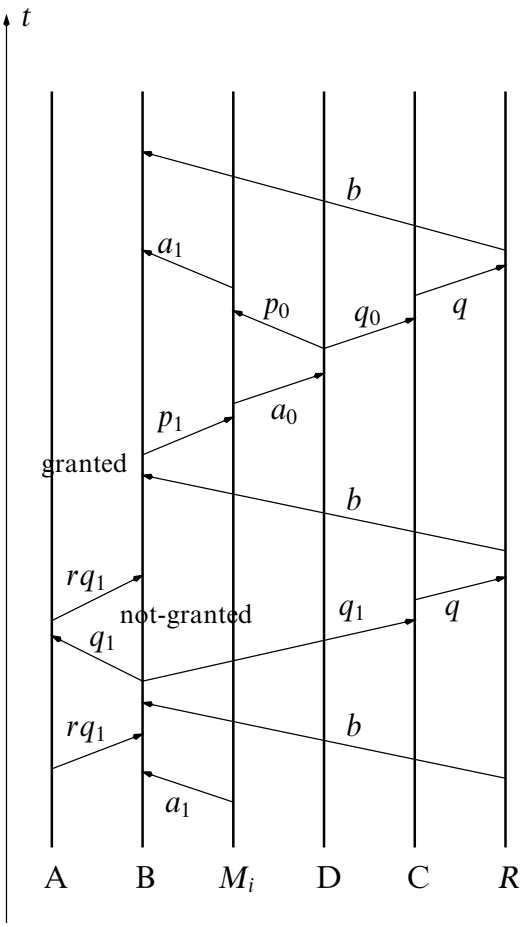

Fig. 14. Space-time graph of the token-ring interface

6. Conclusion

Without giving definite answers to the problems raised in the introduction, we have, nevertheless, tried to provide some hints as to the essential role played by physical time in computation. The necessary reference to physical time in physical implementations of logical circuits forces one to give an explicit treatment of computation interferences. These arise as obstructions when trying to make the causality of underlying logic circuits coincide with the physical causality of their implementations. For synchronous circuits, these may be avoided by ruling the whole circuit with a single clock, which thus provides a global reference to a Newtonian time. In general, however, circuits must be considered as asynchronous and physical space-time as relativistic. In the latter, not all points are causally related, but only those for which one point falls within the light cone issuing from the other. In this respect, asynchronous circuits and relativistic space-time share the same founding point of view. Points derive from events and not the converse, propagating

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

events being treated as primary entities and not as successions of points. The distinction between two classes of points can also be seen in a simple way: two points are causally related if and only if there exists a path between them using vertical or propagation lines (in different spatial directions, but in the same time direction); on the other hand, points defined on two different events originating from the same point are not causally related (Russell 1927). Similarly, for a concurrent computation, each computing path connects points that are causally related. Avoiding computing interferences corresponds to requiring that different computing paths respect the same time ordering, but only for pairs of causally related points.

Remedies for computing interferences in asynchronous circuits consist of recognising paths that may conflict with a module specification, and in possibly delaying these paths so that they respect a prescribed time ordering. This can be done either physically, at the implementation level by introducing explicit time delays, or at the logical level, by imposing specification rules that prevent the occurence of such conflicts. The latter solution, by imposing delay insensitivity both on circuit specification and decomposition in a consistent way, has the advantage of providing a purely logical characterisation of the causal constraints. DI circuits then build a class that may be seen as intermediate between synchronous circuits and general asynchronous circuits. They share with the former the possibility of being completely characterised by formal expressions and rules. But they rely on the same causal structure as the latter. Synchronous circuits rely on time simultaneity classes, and thus on a causal structure that is typical of Newtonian space-time. Asynchronous circuits, on another hand, rely on a consistent treatment of propagation delays and time ordering, and hence on a causal structure that characterises relativistic space-time.

Delay-insensitivity provides an interesting transition between local properties, such as those defining sequential processors, and global ones, such as those exhibited by distributed systems. But DI circuits hardly exhaust the computation possibilities brought by the introduction of asynchronous circuits. The critical consequences of delay sensitivity suggest we consider a further alternative when attempting to classify the different types of computations, that is, those performed by synchronous, by DI asynchronous and by DS (delay-sensitive) asynchronous circuits. Similarly, in the same way as asynchronous computing machines may not always allow simulation by synchronous computing machines, one may infer that physical processes and physical laws, which intrinsically obey relativistic causality, may be simulated by synchronous machines in particular cases only. This hints at another advantage of computations based on asynchronous circuits, that is, the ability to simulate real physical processes in a universal way.

References

Chakraborty, S., Yun, K. Y. and Dill, D. L. (1988) Practical timing analysis of asynchronous circuits using time separation of events. *Proceedings of the IEEE Custom Integrated Circuits Conference*, IEEE Computer Society Press (May) 455–458.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

Davis, A. and Nowick, S. M. (1995) Asynchronous circuit design: Motivation, background and methods. In: Birtwistle, G. and Davis, A. (eds.) *Asynchronous Digital Circuit Design, Workshop in Computing*, BCS-Springer 1–49.

Ebergen, J. C. (1987) *Translating Programs into Delay-Insensitive Circuits*, thesis, Technische Universiteit Eindhoven.

Ebergen, J. C. (1991) A formal approach to designing delay-insensitive circuits. *Distributed Computing* **5** 107–119.

Einstein, A. (1905) Zur Elektrodynamik bewegter Körper. *Annalen der Physik* **17** 891–921.

Furber, S. (1995) Computing without Clocks: Micropipelining the ARM Processor. In: Birtwistle, G. and Davis, A. (eds.) *Asynchronous Digital Circuit Design, Workshop in Computing*, BCS-Springer 211–262.

Guinot, B. (1997) Application of general relativity to metrology. *Metrologia* **34** 261–290.

Hoare, C. A. R. (1978) Communicating Sequential Processes. *Communications of the ACM* **21** (8) 666–677.

Jaekel, M. T. and Reynaud, S. (1998) Quantum observables associated with Einstein localisation. In: Zavattini, E., Bakalov, D. and Rizzo, C. (eds.), *Frontier Tests of QED and Physics of the Vacuum (Sandansky, 1998)*, Heron Press 389–404.

Lamport, L. (1978) Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* **21** (7) 558–565.

Landau, L. D. and Lifschitz, E. M. (1970) *Physique Theorique, tome II: Theorie des Champs*, MIR, Moscou.

Leschiutta, S. (1991) Time Synchronization using Laser Techniques. In: *Special Issue on Time and Frequency: Proceedings of the IEEE* 1001–1008.

Lewandowski, W. and Thomas, C. (1991) GPS Time Transfer. In: *Special Issue on Time and Frequency: Proceedings of the IEEE* 991–1000.

Mallon, W. C., Udding, J. T. and Verhoeff, T. (1999) Analysis and applications of the XDI model. In: *Async'99, Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press 231–242.

Martin, A. J. (1986) Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed computing* **1** 226–234.

Martin, A. J. (1990a) Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In: Hoare, C. A. R. (ed.) *Developments in Concurrency and Communication*, Addison-Wesley 1–64.

Martin, A. J. (1990b) The Limitations to Delay-Insensitivity in Asynchronous Circuits. In: Dally, W. J. (ed.) *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, MIT Press 263–278.

Martin, A. J., Lines, A., Manohar, R., Nyström, M., Penzes, P., Southworth, R., Cummings, U. and Tak-Kwan-Lee (1997) The design of an asynchronous MIPS R3000 microprocessor. In: *Proceedings of the Seventeenth Conference on Advanced Research in VLSI*, IEEE Computer Society Press 164–181.

Matherat, P. and Jaekel, M. T. (1996) Dissipation logique des implémentations d'automates – dissipation du calcul. *Technique et science informatiques* **15** (8) 1079–1104. (English translation at the URL `http://www.arxiv.org/abs/quant-ph/9805018`.)

Mazurkiewicz, A. (1989) Basic Notions of Trace Theory. In: de Bakker, J. W., de Roever, W. P. and Rozenberg, G. (eds.) Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency. School/Workshop, Noordwijkerhout, The Netherlands, May-June 1988. *Springer-Verlag Lecture Notes in Computer Science* **354** 285.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press

Molnar, C. E., Fang, T. P. and Rosenberger, F. U. (1985) Synthesis of Delay-insensitive Modules. In: *1985 Chapel Hill Conference on VLSI* 67–86.

Newton, I. (1687) *Principia*, University of California Press, 1962. (First edition *Philosophiae Naturalis Principia Mathematica* 1687.)

Petit, G. and Wolf, P. (1994) Relativistic theory for picosecond time transfer in the vicinity of the Earth. *Astronomy and Astrophysics* **286** 971–977.

Quinn, T. J. (1991) The BIPM and the Accurate Measurement of Time. In: *Special Issue on Time and Frequency: Proceedings of the IEEE* 894–905.

Russell, B. (1900) *The Philosophy of Leibnitz*, Routledge, London, 1997. (First edition 1900.)

Russell, B. (1925) *ABC of Relativity*, Routledge, London, 1993. (First edition 1925.)

Russell, B. (1927) *The Analysis of Matter*, Routledge, London, 1992. (First edition 1927.)

Seitz, C. L. (1980) System timing. In: *Introduction to VLSI Systems*, Addison-Wesley, Chapter 7 218–262.

Shannon, C. E. (1938) A Symbolic Analysis of Relay and Switching Circuits. *AIEE Transactions* **57** 713–723.

Stevens, K., Ginosar, R. and Rotem, S. (1999) Relative timing. In: *Async'99, Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press 208–218.

Sutherland, I. E. (1989) Micropipelines. *Communications of the ACM* **32** (6) 720–738.

Udding, J. T. (1986) A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing* **1** 197–204.

Vessot, R. F. C. (1991) Applications of Highly Stable Oscillators to Scientific Measurements. In: *Special Issue on Time and Frequency: Proceedings of the IEEE* 1040–1053.

Winskel, G. (1989) An Introduction to Event Structures. In: de Bakker, J. W., de Roever, W. P. and Rozenberg, G. (eds.) Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency. School/Workshop, Noordwijkerhout, The Netherlands, May-June 1988. *Springer-Verlag Lecture Notes in Computer Science* **354** 364–397.

Wolf, P. and Petit, G. (1995) Relativistic theory for clock syntonization and the realization of geocentric coordinate times. *Astronomy and Astrophysics* **304** 653–661.

Wrinkler, G. M. R. (1991) Synchronization and Relativity. In: *Special Issue on Time and Frequency: Proceedings of the IEEE* 1029–1039.

https://doi.org/10.1017/S0960129503004067 Published online by Cambridge University Press