Modeling and Design of Asynchronous Circuits

MARK B. JOSEPHS, STEVEN M. NOWICK, AND C. H. (KEES) VAN BERKEL, MEMBER, IEEE

This technology review explores the behavioral and structural design domains for asynchronous circuits and systems. It proceeds bottom up, introducing relevant concepts, terminology, and techniques through a succession of simple examples. There are seven main points.

1) *Signal transitions provide a key to understanding the switching behavior of asynchronous logic.*
2) *Burst-mode circuits and speed-independent control circuits offer reliable operation that is free from glitches.*
3) *Various notations are available for specification of control circuitry and as a starting point for logic synthesis.*
4) *Bundled data and delay-insensitive coding schemes are suitable for representing data because they address the issues of data validity and completion detection.*
5) *Asynchronous networks, constructed from modules and channels, provide a systems architecture for asynchronous design.*
6) *Handshaking on channels, which controls data communication and synchronization between modules, is implemented using signal transitions.*
7) *The translation of algorithmic descriptions into asynchronous networks facilitates an automated approach to large-scale system design.*

Keywords— *Asynchronous circuits, bundled data, burst-mode circuits, delay-insensitive codes, handshaking, hazards, self-timed circuits, signal transitions, speed-independent circuits, State Graphs.*

I. INTRODUCTION

We are concerned with the behavior and structure of digital systems, whether of microprocessors, digital signal-processor (DSP) cores, application-specific integrated circuits (ASIC's), or individual submodules. The transfer of data between registers in such systems needs to be regulated so as to ensure that data are current and valid whenever they are processed. The standard approach is to synchronize the entire system to a common periodic signal (the clock). Asynchronous, or self-timed, design contrasts with this approach in that the task of regulation is devolved to local control signals [11], [40].

In the same way that the rising and falling edges of a clock start or stop the flow of data through latches in a synchronous circuit, the transitions (not simply the levels) of local control signals regulate activity in an asynchronous circuit. In Section II, we shall look at the behavior of logic gates, in terms of how they propagate signal transitions. State Graphs will be introduced as a convenient notation for modeling this switching behavior.

Complex control functions can be realized by combining gates into burst-mode circuits or speed-independent (SI) circuits (Section III). These two classes of circuits differ in how their correct operation depends upon the relative delays along different signal paths.

When designing control circuitry, one begins by writing a specification. A variety of notations have been devised for that purpose and a selection of these will be illustrated in Section IV. Synthesis takes us from specification down to logic implementation. In Section IV, we shall also consider some of the automated computer-aided design (CAD) tools currently available.

Data processing is our concern in Section V. The problem here is to determine when the processing task has completed. We introduce and contrast the two approaches of data-bundling and delay-insensitive coding.

At a higher level of abstraction, asynchronous systems can be constructed out of modules (engaging in control and data-processing functions) and channels. Handshaking as the means of communication and synchronization between modules is central to this systems architecture, as is discussed in Section VI. Moreover, high-level algorithmic descriptions can be automatically translated into such asynchronous networks.

II. SWITCHING BEHAVIOR OF LOGIC GATES

Digital logic design is concerned with the processing of signals with discrete values, the logic levels zero and one. The basic processing element is the logic gate, which computes an output signal as a function of input signals. For example, the function computed by an OR gate with two inputs, named a and b, and an output, named z, is

Manuscript received September 21, 1998. This work was supported by the European Commission under Working Group 21949 ACiD-WG as part of the ESPRIT Fourth Framework and by the National Science Foundation under Grants MIP-9501880 and CCR-97-31803.

M. B. Josephs is with the Center for Concurrent Systems and VLSI, School of CISM, South Bank University, London SE1 0AA U.K.

S. M. Nowick is with the Department of Computer Science, Columbia University, New York, NY 10027 USA.

C. H. van Berkel is with Philips Research Laboratories, Eindhoven 5656 AA The Netherlands, and with Eindhoven University of Technology, Eindhoven 5600 MB The Netherlands.

Publisher Item Identifier S 0018-9219(99)00879-8.

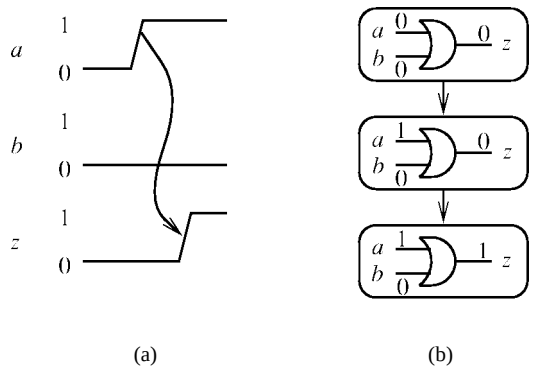

Fig. 1. Equivalent representations of the switching of an OR gate. (a) Timing diagram, with the dependency of the output transition on the input transition indicated. (b) Sequence of three snapshots, each showing the input and output signals of the gate.

defined by the Boolean equation $z = a + b$ (sometimes written $z = a \vee b$).

An understanding of the switching behavior of logic gates is fundamental to an appreciation of asynchronous circuit design. To gain this understanding, one needs to think in terms of signal transitions, i.e., changes in level, either from zero to one or from one to zero. Fig. 1 shows two representations of a switching sequence involving an input transition followed by an output transition.

The transition of a signal x can be denoted by $\tilde{x}$ or simply by x itself. Alternatively, an up-going transition can be denoted by $x\uparrow$ or by x^+, and a down-going transition by $x\downarrow$ or x^-. A Boolean equation translates directly to a set condition, which determines when an up-going transition can occur. Its complement, the reset condition, determines when a down-going transition can occur. For example, the set and reset conditions for an OR gate might be written $S_z = a+b$ and $R_z = a' \cdot b'$, or $a \vee b \mapsto z\uparrow$ and $\bar{a} \wedge \bar{b} \mapsto z\downarrow$ [28].

The class of logic gates can be extended to allow state-holding operators, in which case the reset condition can be stronger than the complement of the set condition. For example, the commonly-used *C-element* [32] is defined by the Boolean equation $z = ab + z(a + b)$. Its output z is set to one when both inputs are one, it is set to zero when both inputs are zero, and it holds its state when the logic levels of a and b differ. More formally, the set and reset conditions can be written as $a \wedge b \mapsto z\uparrow$ and $\bar{a} \wedge \bar{b} \mapsto z\downarrow$, respectively.

A (binary-coded) State Graph is a form of finite-state machine [12]. It can be derived systematically from the set and reset conditions of a logic gate and provides a more explicit model of the switching behavior of the gate. A logic gate with n inputs and one output can be modeled by a machine with 2^{n+1} states. A state is stable if: 1) the set condition is met, but the output is already at logic one; 2) the reset condition is met, but the output is already at logic zero; or 3) neither set nor reset condition is met. In an unstable state, i.e., when the set condition is met and the output is at logic zero, or when the reset condition is met and the output is at logic one, the output is said to be excited. For example, the State-Graph models (Fig. 2) of an

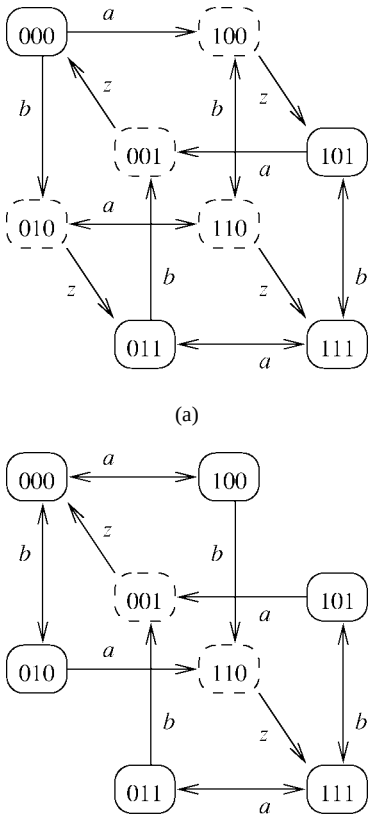

Fig. 2. State-Graph models: (a) OR gate and (b) C-element. Each state (vertex of the graph) is labeled with the value of the vector abz. Unstable states are dashed. Each signal transition (edge of the graph) is labeled with the name of the signal engaging in that transition. Transitions that may be unsafe have been omitted.

OR gate and of a C-element both have 2^3 =8 states, with 000 one of the stable states and 110 one of the unstable ones.

The following three rules govern the switching behavior of logic gates.

1) If a gate is in a stable state, then it will remain in that state until one (or more) of its inputs is transitioned. For example: a^+ would cause the OR gate or C-element to move from state 000 to state 100; concurrent transitions a^+ and b^+ would cause them to move from 000 to 110—in Fig. 2 the transitions are modeled as occurring one after the other (000 $\rightarrow$ 100 $\rightarrow$ 110 or 000 $\rightarrow$ 010 $\rightarrow$ 110).
2) If a gate is in an unstable state and its inputs are maintained at their current levels, then eventually the gate will transition its output. For example, from state 001 the OR gate above would cause z^- to take place and enter state 000.
3) If a gate is in an unstable state, then an input transition is considered to be safe only if its output will remain excited. For example, b^+ is safe when the OR gate, Fig. 2(a), is in state 100; indeed b^+ and z^+ can take place concurrently. Otherwise, the transition is unsafe because it could result in a glitch—this is sometimes

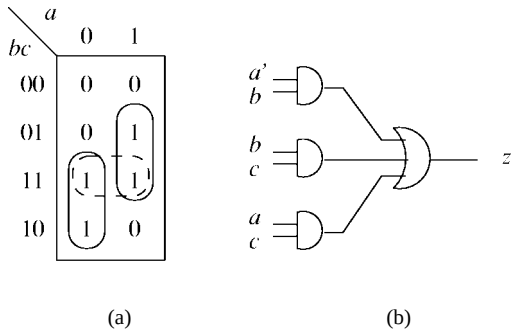

Fig. 3. Logic hazard elimination. (a) "Static-1 hazard" observed in Karnaugh map. (b) Redundant product bc is added. Without the additional AND gate, a glitch might occur on z following the transition of a, as one input to the OR gate rises and the other falls.

called computation interference, a conflict situation, or simply a hazard. For example, a^- may be unsafe in state 100 (an unstable state of the OR gate) because 000 is a stable state.

III. Combining Gates into Circuits

We can obtain reliable circuits by combining gates so as to avoid glitches. Such circuits are called "hazard free" or "semi-modular," depending on whether they are designed to operate in "fundamental mode" or in "input–output mode," as we shall see. Furthermore, we can distinguish between the primary outputs of a circuit, which must be free from glitches, and any internal signals, which need not.

The classic problem is to realize a Boolean function as a sum of products, i.e., in hazard-free two-level logic. The fundamental mode of operation is assumed, viz., the circuit will be given time to settle in a stable state before inputs are allowed to change. In the simplest case, only one input transition is allowed at a time (single-input-change). We can solve the problem by drawing a Karnaugh map—there is no need to construct a State Graph—and either adding "redundant" AND gates, or selecting an alternative set of AND gates (i.e. logic cover), to avoid "static-one hazards," e.g., Fig. 3 solves the problem for $z = a' \cdot b + a \cdot c$.

A generalization of the problem (burst-mode operation [10], [34]) allows specified multiple-input changes (concurrent input transitions) to occur when the circuit is in a given stable state.

Unfortunately, it is impossible to implement reliably certain functions with any logic circuit. For example, we can see from the K-map of Fig. 3 that, if a and b are both allowed to transition in state $abcz = 0000$, the output change may be enabled, then immediately disabled. This is because b may change first, causing the circuit to pass through state 0100 (in which z is excited) before entering state 1100. Such an input change is said to have a "function hazard." The same problem arises, in different guises, in both burst-mode (function hazards) and SI methods (violation of semi-modular constraints). Fortunately, starting from most real-world specifications, function hazards are usually avoidable: they tend to represent irregular behaviors (i.e., disabling an output before processing it) that do not arise in practice. Still, synthesis methods typically must ensure that function hazards are not introduced as a by-product of the various synthesis steps.

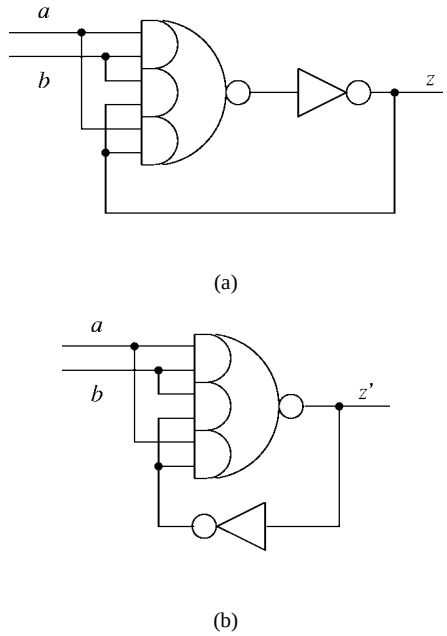

Fig. 4. Implementation of C-element. (a) SI circuit. (b) Burst-mode circuit.

Given an input change which is function-hazard-free, a further generalization of the problem is to realize several functions together (multiple-output functions), which allows the implementation to share product terms. Tools, such as HFMIN [14] and IMPYMIN [39], are available that automatically and efficiently perform hazard-free two-level logic minimization.

In combining gates into circuits we often make use of feedback loops. Fig. 4 illustrates two possible implementations of the C-element using an AND-OR-NOT gate and an inverter. The first implementation is designed to function correctly no matter what the speeds of the two gates, also called an SI circuit. Unlike burst mode, these circuits typically operate in input–output mode: new inputs can be accepted as soon as an output is generated (i.e., no fundamental mode timing assumptions). An important requirement of SI circuits is the "isochronic fork" assumption: whenever a signal transition is forked to several gates, the wire delays on each fork must be equal. The implementation of isochronic forks in very large scale integration (VLSI) needs some care [2], [27].

The second burst-mode implementation assumes that the circuit is in a stable state before each new burst of input transitions arrives. (The danger is that a single transition a or b might occur while the inverter is excited, resulting in a premature transition z'.)

In this example, the burst-mode implementation may enjoy a speed advantage because an inverter delay is removed from the critical path. Whether or not it does so depends on the load on z'; also, the output signal of one implementation is the complement of the output signal of the other. So which implementation is preferable depends

on the context. (Fig. 4 does not show the initialization circuitry that is sometimes needed for a C-element.)

In general, SI circuits are more robust, portable, and easier to verify than circuits that rely on timing assumptions (e.g., relative speeds of gates and delays in the environment). The disadvantage is that they may be larger and slower. Burst-mode circuits have very robust combinational circuitry: the circuits are guaranteed hazard free under all possible gate and wire delays (without any isochronic fork assumptions). However, for correct sequential operation, burst-mode requires two constraints to be satisfied: 1) generalized fundamental-mode operation (allowing multiple-input changes) and 2) a lower bound on the latency of each feedback path (delays must be inserted into the circuit if these bounds are not met).

Reliable circuits can also be obtained if any glitches that might occur are never propagated to the environment. An example is when a glitch feeds directly into an OR gate and the other input to that gate is at logic one for the duration of the glitch. Fundamental-mode circuits can take advantage of this assumption by relaxing the hazard-freedom requirements on the combinational logic. Other examples are arbiters and synchronizers. These are problematic because it takes an unbounded time for a signal in a metastable region [37] to resolve either to logic zero or logic one. In a synchronous design, we can only hope to reduce the mean time between failure. In asynchronous design, we can take special measures (analog filters) [22], [28] to shield the output signals of arbiters and synchronizers from glitches—e.g., see [23].

IV. Specification and Synthesis of Control Circuits

Designers of asynchronous logic face the usual synthesis problem of determining what control circuitry will implement a given specification. We begin this section by considering a few of the notations that are available for writing specifications.

1) *(Abstract) State Graphs.* Fig. 5(a), for example, specifies a join operator on transitions employing a more abstract kind of State Graph than that used in Section II. The number of states and transitions would double if we wanted to distinguish between up-going and down-going transitions, e.g., by labeling states with a vector of signals, as in Fig. 2. Nevertheless, the indirect representation of concurrent transitions (of signals a and b, in this example) by all their possible interleavings (transition a followed by transition b; and b followed by a) adds to the complexity of such a specification. Translation into a binary-coded State Graph can be carried out automatically once the signal levels of the initial (marked) state have been chosen.
2) *Delay-Insensitive Processes.* A join operator will function correctly even if its inputs and output are subjected to arbitrary delays. Delay-insensitivity implies that if a given sequence of transitions is allowed (e.g., a, b, z), then certain other sequences (e.g., b, a, z) must be allowed too. This makes it possible to write succinct specifications in DI-Algebra [21], e.g., the join operator can be specified by the recursively defined process `J = a?;b?;z!;J`. This process can be translated automatically into the State Graph of Fig. 5(a) [26].
3) *Burst-Mode Specifications.* In this form of graph, the number of vertices is reduced at the cost of a more complex labeling of the edges, e.g., Fig. 5(b). This time vertices represent stable states and edges represent a burst of input transitions causing a burst of output transitions [34].
4) *Signal Transition Graphs (STG's).* STG's [6], like Petri nets, facilitate concise specification by capturing input–output causality. In contrast with State Graphs and Burst-Mode Specifications, it is the vertices of an STG that correspond to signal transitions and the marking of edges that indirectly represents the state. The edges of an STG actually correspond to dependencies between signal transitions [Fig. 5(c)] just as one might show on a timing diagram. A signal transition is possible only when all its incoming edges are marked. To simulate the occurrence of the transition, these markings are removed and all its outgoing edges are marked, representing the change in state.

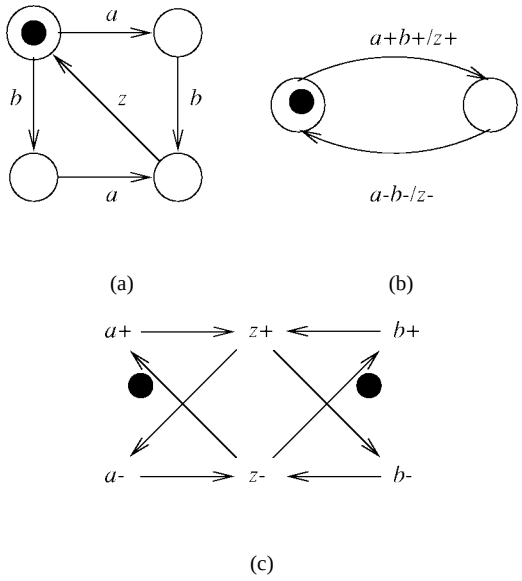

Fig. 5. Equivalent specifications of transition joining. (a) State Graph. (b) Burst-Mode Specification. (c) STG.

Observe that a Muller C-element meets the specification of a join operator. Actually, the C-element is capable of additional functionality that will not be exercised when it fulfills this function, e.g., from state 000 it is safe for a^+ to be followed immediately by a^-.

How do we get from specification to implementation? A number of automatic synthesis and verification tools for asynchronous logic have been developed. For SI circuits, these include tools by Chu [6], Varshavsky *et al.* [42],

Martin [28], van Berkel [3], and Beerel [1], among others. For example, this Special Issue contains a paper [24] on the PETRIFY tool [9] for SI circuits. The tool performs the following steps.

1) Construction of a State Graph from an STG. In order to achieve a consistent state coding, additional signals must be inserted in a way that does not introduce timing dependencies. PETRIFY includes heuristics for signal insertion based on the "theory of regions" [8].
2) Extraction of output and next-state functions from the State Graph, and generation of a set of Boolean equations by logic minimization.
3) Decomposition of high-fanin gates so as to preserve speed independence (SI) [24] and mapping on to a library of gates.

The verification tool AVER for SI circuits described in Dill's Ph.D. dissertation [12] is well known and can be readily used to verify such circuits.

Likewise, many CAD tools and algorithms have been developed for the synthesis and verification of burst-mode circuits. Synthesis tools include the locally clocked method [34], three-dimensional (3-D) method [44], MEAT [10], and ACK [25]. A comprehensive burst-mode CAD environment, called MINIMALIST [15], has recently been introduced.[1] Each of these methods performs the following steps:

1) state minimization and state assignment, using constraints so as to avoid function hazards and critical races [14], [34], [44];
2) hazard-free logic minimization [14], [34];
3) technology mapping [20].

Chakraborty *et al.* have contributed a paper [5] to this Special Issue on an algorithm for timing analysis. This can be used (as illustrated in their paper) to determine timing constraints that ensure the reliable operation of burst-mode circuits. As an example, the tool is applied to circuits produced by the 3-D synthesis tool [44], [45].

V. Representation and Processing of Data

When designing data paths for synchronous systems, a designer trades off clock rate against complexity of each processing stage so as to ensure that data are valid at the beginning and end of each clock period. The issues of data validity and completion detection of processing are handled very differently in asynchronous design, either by careful matching of delays or by special data coding schemes. Both approaches allow processing to take place at the natural frequency of the task, rather than under the worst-case conditions required by a global clocking scheme. Unfortunately, the performance of asynchronous systems is hard to predict; care must be taken to avoid bottlenecks. In this Special Issue, one paper [13] provides a theoretical analysis of the performance of asynchronous pipelines, and another paper [31] describes an experiment which reveals just how fast asynchronous pipelines can be made to perform.

In Section V we examine the two most common approaches to the representation of data that are available to the asynchronous designer. How the choice of representation affects the implementation of data-processing functions will also be illustrated.

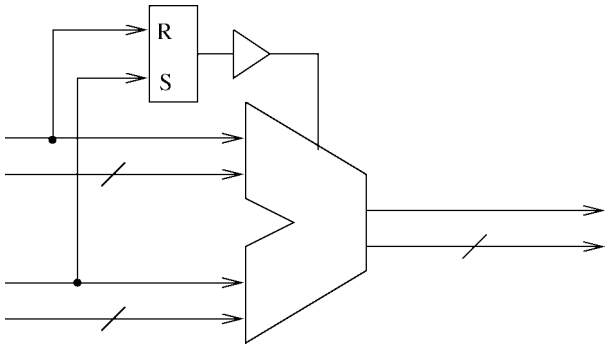

Fig. 6. Merging two bundled-data streams.

A. Data Bundling

The most area-efficient representation of data (in terms of number of wires and, hence, number of gates driving those wires) employs one wire per bit—it is called single rail. In this case, asynchronous designers are free to use standard (i.e., nonhazard-free) implementations of functional units (such as adders). This Special Issue describes several systems [17], [23], [33], [38] that use single rail.

The single-rail representation requires a data-valid signal to be "bundled" with the data wires. It is therefore necessary to detect when processing by a functional unit has completed so that the data-valid signal that accompanies its output data can be asserted. The data-valid signal is typically implemented as a matched delay, designed to equal or exceed the delay of the stage's worst-case computation. Matched delays are typically built out of inverter chains, or else by replicating a critical path of the stage.

As an example, when a multiplexer is used to merge two data streams, the data-valid signals themselves should be multiplexed so as to match the delay of multiplexing each bit. These data-valid signals can also be forked into an set–reset (RS) latch which switches the multiplexer (if necessary) so that it will input from the corresponding data stream (Fig. 6) [35]. (Here we have assumed that at most one of these signal is asserted at any time. Also, in general, data matching requires care to be taken with drive strengths and layout.)

A key advantage of bundled data paths is the use of single-rail area-efficient function blocks. However, a drawback in some applications is that by using a matched delay, the operation is fixed to the worst-case: a stage cannot exhibit data-dependent operation.

[1] [Online.] Available WWW: http://www.cs.columbia.edu/~nowick.

Table 1
Delay-Insensitive Codes for Input Data (Bits a and b) and Output Data (Bits s and c) of a Half Adder

af	at	bf	bt	
0	0	0	0	spacer
1	0	1	0	a=0 & b=0
1	0	0	1	a=0 & b=1
0	1	1	0	a=1 & b=0
0	1	0	1	a=1 & b=1
sf'	st'	cf'	ct'	
1	1	1	1	spacer
0	1	0	1	s=0 & c=0
0	1	1	0	s=0 & c=1
1	0	0	1	s=1 & c=0
1	0	1	0	s=1 & c=1

B. Delay-Insensitive (DI) Coding

A very different approach to the representation of data involves DI codes [43]. These have the attractive property that a receiver is able to determine that a codeword has arrived without relying on any timing assumptions, thus promoting robustness, portability, and ease of design. As a result, a stage can indicate its true data-dependent completion—it is not timed for worst-case operation.

A two-phase (return-to-zero) discipline is usually adopted in which data communication alternates between a working phase and an idling phase [42]. Data changes from spacer (zero state) to proper codeword in the working phase and changes back again in the idling phase.

Consider once again the problem of merging two data streams $a_0 \ldots a_n$ and $b_0 \ldots b_n$ into a third stream $z_0 \ldots z_n$. With a two-phase delay-insensitive representation of data, this can be achieved very simply. It requires an array of OR gates since we want $z_i = a_i + b_i$ for $i = 0 \ldots n$, assuming at least one of the streams is in state $0 \ldots 0$ at any time.

The most popular delay-insensitive code employs two wires (xf and xt, say) per bit (x)—it is called double-rail or dual-rail [37]. If the spacer is represented by both wires at logic zero, then in the working phase either the transition xf^+ occurs (to communicate $x = 0$) or xt^+ occurs (to communicate $x = 1$), leaving the wires in state 10 or 01, respectively. A down-going transition on the same wire takes place in the idling phase. Further delay-insensitive codes can be formed by concatenation of double-rail codewords. Table 1 illustrates two "two-out-of-four" delay-insensitive codes, each formed by concatenation of two double-rail encoded bits, a and b, and s and c, respectively.

A self-timed functional unit [37] ensures that input of a proper codeword precedes output of a proper codeword, and input of a spacer precedes output of a spacer. For its part, the circuit to which such a unit connects must ensure that output of a proper codeword precedes input of a spacer, and output of a spacer precedes input of a proper codeword.

One might think that there is a 100% area overhead on the data path when delay-insensitive coding is used instead of single rail. However, differential (i.e., true and complement) signals are frequently required in processing single-rail data, so the penalty need not be so severe. For example, implementation of a single-rail half-adder in CMOS [Fig. 7(a)] requires at least 12 transistors (four for the NAND gate and eight for the AND-OR-NOT gate), but one may also need a further four transistors (two inverters) to obtain signals a' and b'. On the other hand, a dynamic CMOS implementation [Fig. 7(b)] of a half adder using the delay-insensitive codes of Table 1 can be obtained with 19 transistors by following Martin's technique [29].

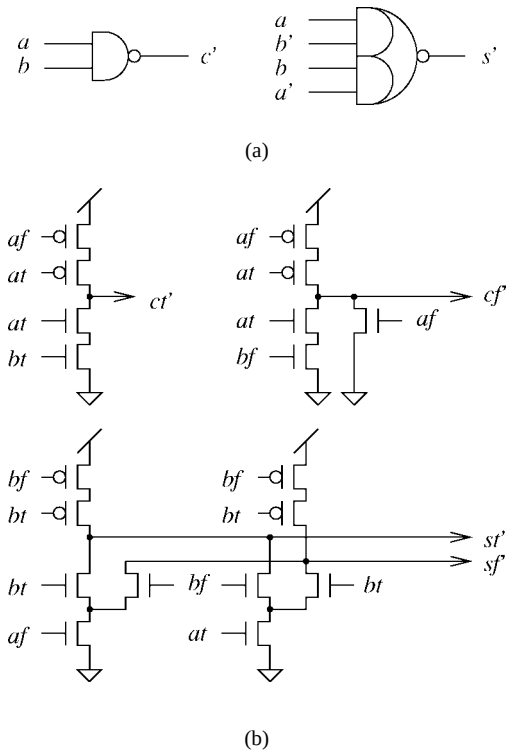

Fig. 7. Half-adder implementations for different data representations: (a) single-rail and (b) delay-insensitive coding.

A key challenge in the design of DI-encoded circuits is to construct an efficient completion-detection circuit. Such a circuit detects when data are valid and is used to signal completion (e.g., to an earlier stage). The challenge is to implement this circuit with minimum overhead; a number of techniques have been developed (see, e.g., [29] and [37]).

As technology continues to shrink, delay-insensitive coding of data may become more attractive than is currently the case. A recent microprocessor design [30] uses two-phase delay-insensitive codes to represent data.

VI. Asynchronous Networks

Asynchronous control and data-processing units can be conveniently packaged into modules that are connected together by channels. Each channel connects a port of one module to a port of another. Synchronization between modules can be accomplished on a channel consisting of two wires, one for requests and the other for acknowledgments. Such systems are called macromodular [7], [41].

Suppose request and acknowledgment on port a are represented by signals ar and ak, respectively, and similarly for port b. A four-phase (return-to-zero) handshake between active port a and passive port b [Fig. 8(a)] then involves the sequence of transitions ar^+, br^+, bk^+, ak^+, ar^-,

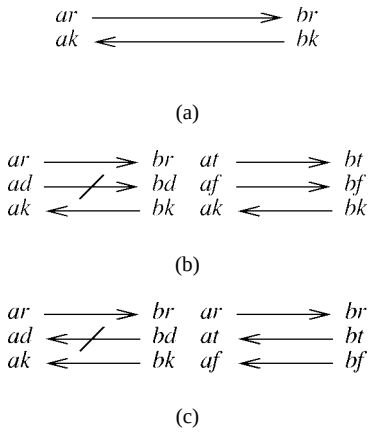

Fig. 8. Channels (port a active, port b passive). (a) Synchronization. (b) Push: (left) single-rail and (right) double-rail. (c) Pull: (left) single-rail and (right) double-rail.

br^-, bk^-, ak^-. Note that handshaking is insensitive to wire delays.

Data communication between modules obviously requires a channel with more wires but can still be accomplished by following a four-phase handshaking protocol. Such channels can be classified as push [Fig. 8(b)] or pull [Fig. 8(c)] according to whether the active port is the producer of data and the passive port the consumer, or vice versa. Consider, for example, a push channel, so that the acknowledgments are generated by the consumer. In the single-rail case [Fig. 8(b, left)] the data-valid signal represents a request whereas in the delay-insensitive case [Fig. 8(b, right)] it is the codeword itself that serves this purpose.

An important module is the pipeline register, which has a passive port on which it captures data and an active port on which it passes data on, i.e., it communicates with other modules through two push channels. A pipeline (Fig. 9) is a linear array of such registers, with modules that perform data-processing functions inserted between registers, as required. Fig. 10(a) shows a simple implementation of a pipeline register for bundled data, using a C-element to close the D-type latches as soon as data have passed through them and to reopen the latches once that data have been assimilated. Although data can flow through an empty pipeline with minimal delay and the level-sensitive latches are simpler than the edge-triggered latches typically used in synchronous pipelines, this implementation only allows 50% pipeline occupancy. With more complex control circuitry, however, it is possible to increase the decoupling between the register's two ports, and in particular to allow a pipeline to fill all its stages [16].

Fig. 10(b) shows another pipeline register implementation, this time for data words consisting of bits in double-rail. The combination of two C-elements and an OR gate serves to latch one bit and generate an acknowledgment signal [32]. Here, we have chosen to "daisy-chain" such units together, e.g., low-order bits are captured and passed before high-order bits. If this matches the order of processing, no loss of performance need result. (If necessary,

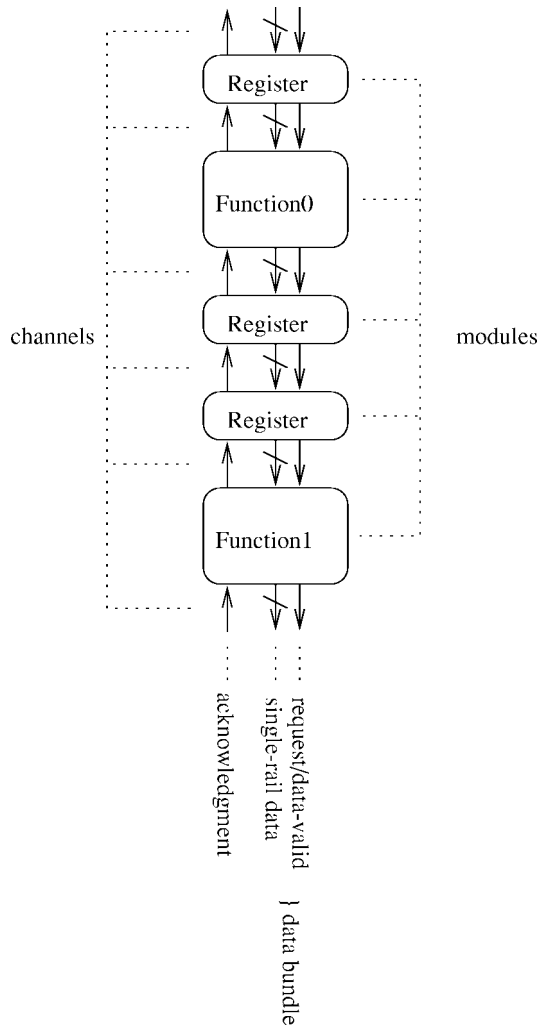

Fig. 9. Pipeline for bundled data.

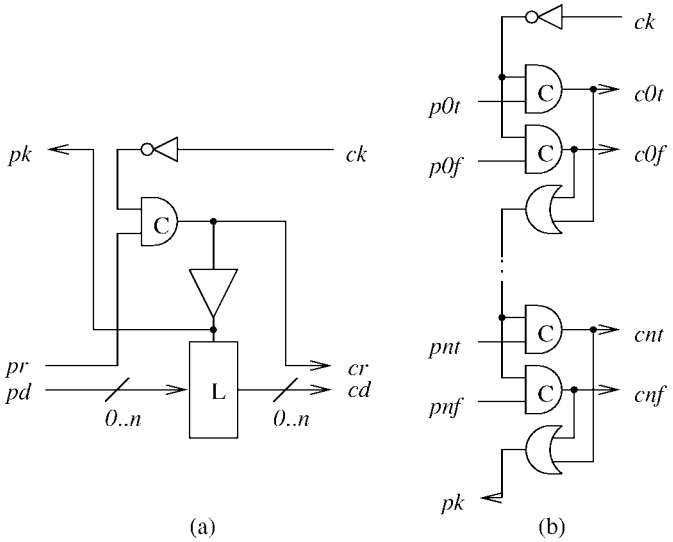

Fig. 10. Pipeline register. (a) Bundled data. (b) Dual rail. Data words are received on port p from a producer module and propagated from port c to a consumer module.

bits can be latched in parallel by forking acknowledgment signals to their respective C-elements and joining the acknowledgments so generated.) To improve on area and energy efficiency, Martin [30] integrates data processing

and latching functions into one module implemented as a single precharge logic block.

Modules can be designed so as to be testable for fabrication faults. For a module to be testable, its input signals have to be controllable and its output signals observable. One of the papers in this Special Issue [36] is devoted to the topic of design-for-test; other papers [17], [23] discuss the testability of actual chips.

A high-level approach to asynchronous design is possible using parallel programming languages in which handshaking is available as a primitive for communication and synchronization, as in CSP [18] or OCCAM. From such algorithmic descriptions, one can apply syntax-directed translation rules to create a network of handshaking modules [3], [4], [28]. In the case of the TANGRAM language, a silicon compiler can generate a handshaking circuit and then a layout entirely automatically. Different back-ends have been developed, allowing a choice of single-rail [35] and double-rail [3] representation of data. Various tools (for functional simulation, fault-coverage and area, power and performance estimation) are available within Philips that facilitate the exploration of the design space at the program level. This Special Issue includes a paper [23] that describes how the TANGRAM tool-set facilitated the design of a standby circuit for a low-power pager.

VII. Conclusion

When designing digital systems, a designer must make high-level decisions concerning systems architecture. When the target circuit-implementation technology is asynchronous, the decisions involve structuring a system as a network of modules that synchronize and communicate with each other by handshaking over channels. A number of papers in this Special Issue are concerned with systems architecture for low power and high performance (microprocessors [17], pagers [23], hearing aids [33], multimedia processors [38], and microprogrammed controllers [19]).

Once the handshaking protocols and data representations employed on channels have been determined, each module can be designed independently of the rest. Through simulation, those modules can be identified that are the bottleneck in system performance and those that are draining too much power or occupying too much area. The design effort can be directed at those modules without having to worry about global system timing.

In order to implement modules, control logic may be separated from the data path, while provision must be made for initialization and testing [36]. If data bundling is used, the data path will look much the same as that in a synchronous implementation. The main differences are likely to be the routing of a data valid signal and the use of latches controlled by local signals, rather than registers controlled by a global clock signal. If double-rail coding is used, the logic blocks on the data path may resemble domino logic [30].

For control logic, the behavior of the module must be defined, perhaps as a burst-mode specification or as an STG. This specification is then used to synthesize a circuit, perhaps an SI circuit using PETRIFY [9] or as a burst-mode circuit using 3D [45] or MINIMALIST [15]. The control logic generated by such tools is guaranteed to be free from hazards. However, any timing assumptions must still be verified, viz., delay margins required for bundled data or burst-mode circuits [5], or isochronic fork assumptions for SI circuits. Of course, timing verification is mandatory in synchronous circuit design.

Alternatively, a system's structure and behavior can be captured in an algorithmic description language, such as TANGRAM, from which an implementation can be generated automatically based on standard choices for handshaking protocol, data representation and circuit timing model. This silicon compilation approach has now been successfully demonstrated on a number of low-power IC's within Philips, e.g., [23].

References

[1] P. Beerel and T. Meng, "Automatic gate-level synthesis of speed-independent circuits," in *Proc. ICCAD'92*, 1992, pp. 581–587.

[2] K. van Berkel, "Beware the isochronic fork," *Integr. VLSI J.*, vol. 13, no. 2, pp. 103–128, 1992.

[3] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming* (International Series on Parallel Computation, vol. 5). Cambridge, U.K.: Cambridge Univ. Press, 1993.

[4] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," in *Proc. ICCAD'89*, 1989.

[5] S. Chakraborty, D. L. Dill, and K. Y. Yun, "Min-max timing analysis and an application to asynchronous circuits," this issue, pp. 332-346.

[6] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga, "A design methodology for concurrent VLSI systems," in *Proc. ICCD'85*, 1985, pp. 407–410.

[7] W. A. Clark and C. E. Molnar, "Macromodular computer systems," in *Computers in Biomedical Research*, vol. IV, R. W. Stacy and B. D. Waxman, Eds. New York: Academic, 1974, ch. 3, pp. 45–85.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Complete state encoding based on the theory of regions," in *Proc. 2nd Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1996, pp. 36–47.

[9] ——, "PETRIFY: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inform. Syst.*, vol. E80-D, no. 3, pp. 315–325, 1997.

[10] A. Davis, B. Coates, and K. Stevens, "Automatic synthesis of fast compact self-timed control circuits," in *Proc. 1993 IFIP Working Conf. Asynchronous Design Methodologies*, Manchester, U.K., pp. 193–207.

[11] A. Davis and S. M. Nowick, "An introduction to asynchronous circuit design," in *Encylopedia of Computer Science and Technology*, vol. 38, supplement 23. New York: Marcel Dekker, 1998.

[12] D. L. Dill, "Trace theory for automatic hierarchical verification of speed-independent circuits," in *ACM Distinguished Dissertations Series*. Cambridge, MA: MIT Press, 1989.

[13] J. Ebergen and R. Berks, "Response time properties of linear asynchronous pipelines," this issue, pp. 308-318.

[14] R. M. Fuhrer, B. Lin, and S. M. Nowick, "Symbolic hazard-free minimization and encoding of asynchronous finite state machines," in *Proc. ICCAD'95*, 1995.

[15] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, and L. Plana, "MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines," in *Proc. IEEE/ACM Int. Workshop Logic Synthesis*, 1998.

[16] S. B. Furber and P. Day, "Four-phase micropipeline latch control circuits," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 247–253, June 1996.

[17] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver, "AMULET2e: An asynchronous embedded controller," this issue, pp. 243-256.
[18] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
[19] H. Jacobson and G. Gopalakrishnan, "Application-specific programmable control for high-performance asynchronous circuits," this issue, pp. 319-331.
[20] K. W. James and K. Y. Yun, "Average-case optimized transistor-level technology mapping of extended burst-mode circuits," in *Proc. 4th Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 70–79.
[21] M. B. Josephs and J. T. Udding, "An overview of DI algebra," in *Proc. 26th Annu. Hawaii Int. Conf. System Sciences*, 1993, vol. 1, pp. 329–338.
[22] M. B. Josephs and J. T. Yantchev, "CMOS design of the tree arbiter element," *IEEE Trans. VLSI Syst.*, vol. 4, no. 4, pp. 472–476, 1996.
[23] J. Kessels and P. Marston, "Designing asynchronous standby circuits for a low-power pager," this issue, pp. 257-267.
[24] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Logic decomposition of speed-independent circuits," this issue, pp. 347-362.
[25] P. Kudva, G. Gopalakrishnan, and H. Jacobson, "A technique for synthesizing distributed burst-mode circuits," in *Proc. IEEE/ACM Design Automation Conf.*, 1996, pp. 67–70.
[26] W. C. Mallon and J. T. Udding, "Building finite automata from DI specifications," in *Proc. 4th Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 184–193.
[27] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proc. 6th MIT Conf. Advanced Research in VLSI*, 1990, pp. 263–278.
[28] ——, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication*, C. A. R. Hoare, Ed. Reading, MA: Addison-Wesley, 1990, pp. 1–64.
[29] ——, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, no. 1, pp. 119–137, 1992.
[30] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee, "The design of an asynchronous MIPS R3000," in *Proc. 17th Conf. Advanced Research in VLSI*, 1997, pp. 164–181.
[31] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland, "Two FIFO ring performance experiments," this issue, pp. 297-307.
[32] D. E. Muller, "Asynchronous logics and application to information processing," in *Switching Theory in Space Technology*, H. Aiken, and W. F. Main, Eds. Stanford, CA; Stanford Univ. Press, 1963, pp. 289–297.
[33] L. S. Nielsen and J. Sparsø, "Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid," this issue, pp. 268-281.
[34] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," in *Proc. ICCAD*, 1991, pp. 192–197.
[35] A. M. G. Peeters, "Single-rail handshake circuits," Ph.D. dissertation, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 1996.
[36] M. Roncken, "Defect-oriented testability for asynchronous IC's," this issue, pp. 363-375.
[37] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds. Reading, MA: Addison-Wesley, 1980, ch. 7.
[38] H. Terada, S. Miyata, and M. Iwata, "DDMP's: Self-timed super-pipeline data-driven multimedia processors," this issue, pp. 282-296.
[39] M. Theobald and S. M. Nowick, "An implicit method for hazard-free two-level logic minimization," in *Proc. 4th Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 58–69.
[40] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley–Interscience, 1969.
[41] S. Unger, "A building block approach to unclocked systems," in *Proc. 26th Annu. Hawaii Int. Conf. System Sciences*, 1993, vol. 1, pp. 339–348.
[42] V. I. Varshavsky, Ed., *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. New York: Kluwer, 1990.
[43] T. Verhoeff, "Delay-insensitive codes—An overview," *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988.
[44] K. Y. Yun, "Synthesis of asynchronous controllers for heterogeneous systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1994.
[45] K. Y. Yun, "Automatic synthesis of extended burst-mode circuits using generalized C-elements," in *Proc. EURO-DAC*, 1996, pp. 290–295.

Mark B. Josephs for a photograph and biography, see this issue, p. 222.

Steven M. Nowick for a photograph and biography, see this issue, p. 221.

C. H. (Kees) van Berkel (Member, IEEE), for a photograph and biography, see this issue, p. 222.