ASYNCHRONOUS LOGIC DESIGN

The performance, cost effectiveness, and complexity of digital computers have experienced explosive growth in the past two decades. These advances in the performance and cost effectiveness of integrated circuits (IC) are a direct result of innovations in the technology for fabrication, architecture, and advanced circuit techniques. Recent advances in semiconductor technology have pushed the number of transistors on a single microprocessor chip beyond the ten million mark and clock frequencies beyond 600 MHz (1). Distributing such a high-frequency clock signal over the entire chip is an extremely challenging task, which is guaranteed to get more difficult with future generations of microprocessors synchronized by a single clock. Thus, it is worth exploring other types of digital design that might offer a solution to this clock distribution problem. Asynchronous logic circuits hold the promise of alleviating these clock distribution problems because they do not require a global clock signal for their operation (2,3).

FUNDAMENTALS AND MOTIVATION

Modern computers are designed with digital logic circuits. The design of digital logic circuits is broadly classified into two basic types: combinational logic design and sequential logic design (4). In combinational logic circuits, outputs depend only on their present inputs. On the other hand, sequential logic circuits are those circuits whose outputs depend on their past as well as present inputs. This implies that sequential circuits must incorporate some form of memory to hold information about past inputs. This information about past inputs contained in the memory elements (also called latches or flip-flops) is called state information. Thus, the output of a sequential circuit, as shown in Fig. 1, is a function of its current input and state. Frequently, sequential circuits are also referred as finite state machines (FSM) or sequential machines.

Sequential circuits can be classified into synchronous or clocked circuits and asynchronous circuits. In synchronous or clocked sequential circuits, time is quantized, and all actions take place at discrete intervals of time determined by a periodic source of pulses called a clock. The clock signal controls the memory elements whose values reflect the circuit state. Inputs to synchronous circuits can change only during the period when the clock pulses essentially disable the memory elements and prevent the circuit from changing state (i.e., events in synchronous circuits are sequenced by a globally distributed periodic clock signal).

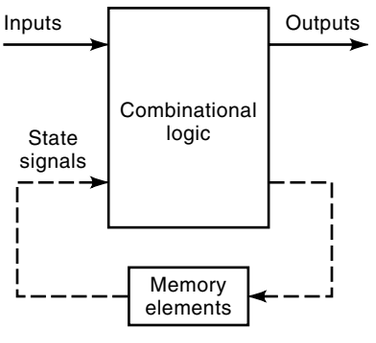

Figure 1. Sequential logic circuit.

The role of a clock in synchronous circuits can be described with an excellent analogy to a conductor in an orchestra regulating every beat of the music (5). In an orchestra, individual performers know exactly when to play each note and know implicitly that their output will blend appropriately with the output of other functional units within the orchestra, so long as they observe the conductor's beat. Unlike an orchestra, synchronous operation is not fundamental to an electronic circuit. The operation of an electronic circuit can be compared to a production line where partly assembled products are passed from one stage to the next when they are ready. A smooth flow must be maintained for efficient production. Asynchronous circuits follow this production line mode of operation.

In contrast to synchronous sequential circuits, asynchronous circuits are a broader class of circuits wherein general events may take place at any time (i.e., they are designed to operate without a global synchronization clock signal). Asynchronous circuits use local synchronization rather than a global synchronous signal such as clock (i.e., various circuit components communicate with each other using handshaking or request-and-acknowledge signals).

Since its advent, digital logic design has been dominated by synchronous logic because of its relative ease of design compared to asynchronous logic circuits. The ease of designing synchronous logic is a direct result of timing restrictions placed on circuit signals and the global clock. These timing constraints ensure that all signals must stabilize before the onset of the clock signal that stores the stabilized signal values into the memory elements. If this constraint is not satisfied, the clock frequency must be reduced to avoid circuit malfunction resulting from the storage of incorrect signal values in the memory elements. In addition, distributing gigahertz frequency clock signals over a complex chip is an extremely challenging task that is guaranteed to get more difficult with future generations of microprocessors synchronized by a single clock. This has recently revived research efforts that eliminate the global clock by employing asynchronous circuits or locally clocked circuits. In addition, asynchronous circuits can also offer the following desirable properties, when designed carefully:

- *Alleviating Global Clock Distribution Problems.* In a synchronous circuit, the clock signal connects all the memory elements and clocked logic gates through wires that are routed over the entire chip. Because long wires have significant resistance and capacitance, electrical signals carried on these wires incur significant delays. Due to this large distributed resistive–capacitive clock network, a given clock pulse arrives at different times at different parts of the circuit. This effect is called *clock skew,* which can severely affect the performance of the circuit. Distributing clock signal over the entire chip with relatively low clock skew is one of the most difficult tasks being faced by the designers of advanced high-performance microprocessors. One of the goals of the clock signal is to synchronize the updating of the state information in the memory elements. This synchronization in the presence of clock skew can result in a circuit malfunction. Because asynchronous circuits do not employ a global clock for

synchronization, the problem of global clock skew and global clock distribution is alleviated. Although asynchronous circuits eliminate the global clock, they replace this problem with synchronization or handshaking constraints at the local level. Thus, a large global problem is replaced by several smaller problems at the local level.

- *Low-Power Potential.* In high-performance microprocessors, designers have developed several methods to address clock distribution and clock-skew issues. Invariably, they involve building a clock distribution tree or mesh with amplifying buffers. Unfortunately, high-performance microprocessor designs with clock frequencies in the 600 MHz range require very large clock buffers to obtain an acceptable clock-distribution network, which results in excessive power consumption and occupy large area. For example, Digital Equipment Corporation's Alpha microprocessor with 433 MHz clock frequency requires a clock-distribution network that occupies 10% of total chip area (16.8 mm $\times$ 13.9 mm) and consumes 40% of total power consumption of 30 W (6). With additional increases in clock frequency, it will be increasingly expensive to distribute a single global clock due to power dissipation constraints. In addition, in a synchronous circuit, the clock signal is always active, even if the circuit is not processing any data. Thus, low-power synchronous circuits usually switch the clock signal off to a subcircuit that is not active. This requires an additional circuit that monitors the input signals for any activity, which itself consumes power. Also, the switch that shuts off the clock signal to an inactive subcircuit presents an additional capacitive load to the clock distribution network. Because asynchronous circuits do not require a continuously running global clock for their operation, they are free of these problems and can provide potential savings in power consumption for low-power applications. Because the market for wireless, portable, and hand-held consumer products that require long battery life and thus low-power circuits is growing very fast, potential low-power benefits are a significant motivating factor in the recent resurgence of interest in asynchronous circuits (7).
- *Improved Performance Potential.* In synchronous sequential circuits, the pulse period of the clock signal is chosen according to the worst-case timing path through the combinational logic. This clock period is quite inflexible in taking advantage of the best-care or even average-case behavior. Because components in asynchronous circuits synchronize events by generating local completion signals when the computation is complete, their performance is not determined by worst-case delays. Thus, asynchronous circuits hold the potential of achieving increased performance.
- *Modularity and Upgradability.* An important aspect of any design methodology is its support for "modularity," since a divide and conquer approach is central to any complex design task. A clock signal is, in effect, a global system variable, and therefore an impediment to modularity. Because asynchronous circuits do not impose a global clock constraint, a system designed with asynchronous components will function correctly by simply connecting these components together (provided their interfaces match and they observe the same signaling protocol). Early research work by Clark and Molnar (7a) demonstrated the composition benefits of asynchronous circuit modules and formed the basis for several recent asynchronous synthesis approaches (54,55). The formation of Virtual Socket Interface (VSI) alliance by leading semiconductor corporations provides significant thrust to the design methodologies that naturally support more modular systems with easy-to-replace components. Asynchronous design methodologies are bound to benefit from this recent push toward modularity and reusability. In synchronous sequential circuits, various components are synchronized using the same global clock signal. Thus, even in the case of minor change in circuit functionality, one must follow the fixed clock period constraint to avoid a major redesign. In contrast, a logic change in asynchronous circuits can be locally accommodated, since various components do not have to adhere to a strict global timing constraint.
- *Environmental Robustness.* Synchronous circuits are designed for correct operation within a range of temperature, power-supply voltage variations, and fabrication process variations. While designing synchronous circuits, the clock time period is chosen according to the delay of the components with worst-case parameters (e.g., maximum allowable temperature and minimum allowable power-supply voltage). Thus, if a synchronous circuit was designed to operate at a maximum temperature of 95 °C, it may not function correctly above this temperature (because of an increase in component delays with increasing temperatures). In contrast, asynchronous circuits are robust with respect to variations in their environment, such as temperature, power supply, and fabrication process because their operation is independent of component delays and thus their variations. In addition, asynchronous circuits offer the advantage of reducing electromagnetic interference (emi) emissions because they distribute the switching energy over time and frequency, in contrast to synchronous circuits where the switching energy is concentrated around clock frequency (5).

In spite of all these potential benefits, asynchronous circuits have not been widely used in commercial applications except where unavoidable, such as interface circuits between two independently clocked circuits or real-time circuits. Even in the case of asynchronous interface circuits, their design is considered to be very informal and is thought to be one of the most difficult tasks by integrated circuit designers. The major reason for the overwhelming popularity of synchronous circuits is their ease of design, which is due to a clear separation between functionality and timing in synchronous designs. This separation is taken away in asynchronous circuits [e.g., in isochronic fork assumption (51) or fundamental mode assumption (15)], which makes the design of these circuits more complex. Although asynchronous circuits eliminate the global clock, they replace this problem with synchronization or handshaking constraints at the local level. Unfortunately, in some cases the overhead due to handshaking can actually result in a performance degradation. To reduce this performance penalty, architectural changes to hide this handshaking overhead must be incorporated (63). In addition, asynchronous circuits are prone to generating unwanted sig-

nal changes known as hazards that may cause a circuit to malfunction. The hazards are naturally filtered out in a synchronous design by choosing a long-enough clock period, which ensures that the circuit is in a stable circuit state before the next input changes take place. Thus, high-performance asynchronous logic circuits that are free of hazards are more difficult to design than their synchronous counterparts. In general, the presence of hazards in asynchronous circuits is a major hindrance in their widespread use. Although an asynchronous implementation of a DCC error corrector chip at Philips Research Labs (72) demonstrated a five times reduction in power over the best synchronous design, a clear advantage of asynchronous designs for large-scale high-performance and low-power circuits still remains to be seen. This chip was designed with the help of an internal asynchronous synthesis tool called TANGRAM. Unavailability of such industrial strength synthesis tools through commercial CAD vendors is another hurdle in the proliferation of commercial asynchronous designs.

Despite few current commercial applications of asynchronous circuits, it is worth exploring other areas of digital design that may offer a solution to the clock distribution and power dissipation problems in high-performance circuits. Asynchronous logic circuits hold the promise of solving these problems, which is a major motivating factor in recent resurgence of interest in asynchronous and locally clocked circuits. Asynchronous design should not be viewed as a single alternative to synchronous logic design. More accurately, synchronous design is a special case representing a single design point in a multidimensional asynchronous design space. Thus, one can implement a logic circuit using completely asynchronous design, or a completely synchronous design, or a suitable combination of the two design techniques.

ASYNCHRONOUS LOGIC

There are many flavors of asynchronous logic. However, there are a few key features that characterize the underlying implementation of various asynchronous designs. These implementation features can be seen as a choice between different synchronization protocols (i.e., two-phase versus four-phase signaling protocol), data encoding schemes (i.e., dual-rail data encoding versus bundled data encoding), and delay models (8).

Asynchronous Control Protocols

In general, asynchronous circuit components communicate with each other using handshaking or request-and-acknowledge signaling protocols. The correct functionality of an asynchronous circuit depends upon each component following a sequence of request-and-acknowledge events in its signaling protocol and is independent of signal timings. One component, the sender, sends a request to start an event to another component, the receiver. After the receiver completes the requested action, it sends an acknowledge signal back to the sender to complete the protocol. In order for this protocol to function correctly, the sender must not produce a new request signal until the previous request signal has been acknowledged. In addition, the receiver must not reply with an acknowledge signal unless it has received the request signal. This standard request acknowledge signaling protocol can be realized using a transition sensitive circuit or a level sensitive circuit, described as follows.

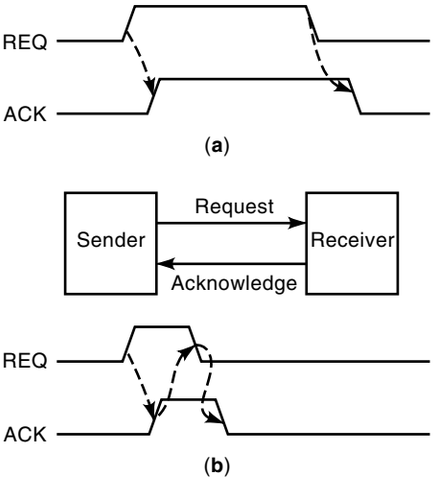

Figure 2. (a) Two-phase signaling; (b) four-phase signaling.

Two-Phase Signaling. In two-phase signaling, also known as transition or nonreturn to zero signaling, the request-and-acknowledge events are actually transitions (i.e., either rising or falling). A request or acknowledge event is said to have occurred if the corresponding signal wire has made a transition from its present state to another (i.e., from high level to low level or vice versa). The direction of the transition is unimportant. As shown in Fig. 2(a), in the two-phase signaling protocol, the sender can issue a request to start an event by generating a transition on the request wire, and the receiver can respond after completing the requested event by generating a transition on the acknowledge wire. There are two states in this signaling protocol: a state where the sender is active and the receiver has not yet responded and a state where the receiver has responded and is waiting for the sender to become active again. In general, both request and acknowledge wires are initialized to level zero. Thus, the inactive state in the protocol is defined as the state when both request and acknowledge wires are at the same level.

Four-Phase Signaling. In the four-phase signaling protocol, also known as return to zero signaling, the request-and-acknowledge events are levels instead of transitions [i.e., either low (zero) or high (one)]. Similar to two-phase signaling, the inactive state in this signaling protocol is also initialized to level zero for both request and acknowledge wires. As shown in Fig. 2(b), a four-phase signaling protocol sequence starts by both request and acknowledge wires at level zero. Then sender initiates a request to start an event by changing the request wire to level one. After completing the requested event, the receiver responds with the acknowledge signal by changing the acknowledge wire to level one. When the sender receives the acknowledge event, it changes the level of request wire back to zero, and subsequently the receiver also changes the level on acknowledge wire back to zero. Thus, after completing the four-phase signaling, both request and acknowledge wires return back to level zero. Now, the sender is ready to send another request signal.

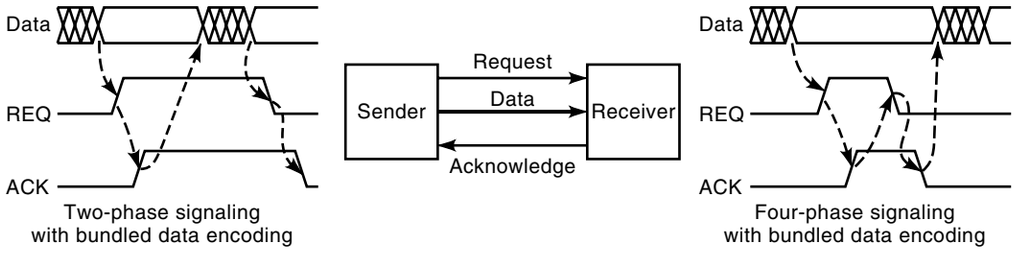

Figure 3. Bundled data encoding.

The choice of either the two-phase signaling or four-phase signaling depends on designers and a particular application. In theory, two-phase signaling may appear to be faster and less power-consuming than four-phase protocol since a complete sequence of two-phase protocol requires half the number of transitions of a four-phase protocol. However, two-phase signaling requires special logic to detect transitions rather than levels that result in more complex circuits. This in turn penalizes the interface in terms of both performance and power.

Asynchronous Data Encoding Schemes

In addition to the choice of two- or four-phase control signaling protocol discussed earlier, the data signals can also be encoded using either bundled data encoding or dual-rail encoding, which is described as follows.

Bundled Data Encoding. Bundled data encoding employs separate wires for data and control signals. As shown in Fig. 3, for n bits of data to be communicated from the sender to the receiver, bundled data encoding requires $n + 2$ wires: n wires for the data, one wire for the request signal, and one wire for the acknowledge signal. Bundled data encoding can be employed with either the two-phase signaling or the four-phase signaling protocol, as illustrated in Fig. 3.

Separation of data and control timing in bundled data encoding imposes certain constraints described as follows. Suppose that the request signal were faster than at least one data signal. In such a case, the receiver would receive the request event from the sender even before the data are received and may initiate the required computation with wrong data values. Thus, the use of bundled data encoding implies an implicit assumption that the request signal is slower are asserted). This constraint is known as bundled data constraint and is widely used in the design of asynchronous logic circuits. Thus, request signal in bundled data encoding is similar to a clock signal.

Dual-Rail Encoding. In the case that the bundled data constraint cannot be satisfied, dual-rail encoding (shown in Fig. 4) is used where data and request signals are encoded together onto two wires (dual rails). In addition, an acknowledge wire is also required for every data bit. Thus, communicating n data bits with this encoding requires $2n$ wires for the dual-rail encoding of request and data signals and n wires for acknowledge signals. In the case that the receiver requires the availability of all data bits before activating acknowledge signal, only one acknowledge wire is sufficient for all dual-rail data signals, which reduces the total number of wires from $3n$ to $2n + 1$. In general, the values on the dual rails are interpreted as follows: 10 corresponds to data value 0, 01 implies a data value 1, 00 implies that the data value is not yet available, and 11 is not allowed. Dual-rail encoding is insensitive to the delays on any wire and is often used when bundled data constraint cannot be satisfied. Unfortunately, it results in increased complexity of both the number of wires and the logic. To compensate for the high complexity of dual-rail encoding, other coding schemes have been developed (8a,8b) in which the logic overhead is only in the completion detection and the complexity of data processing is avoided.

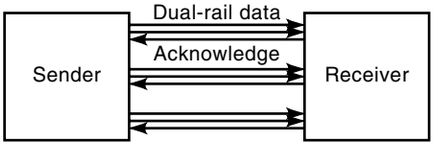

Figure 4. Dual-rail data encoding.

Delay Models and Hazards in Asynchronous Circuits

A formal design method for a digital circuit requires a delay model of the logic gate operation. The accuracy of this model determines the practicality of the design method. Logic gates and interconnections (i.e., wires are fundamental building blocks of digital circuits). Physically, both logic gates and wires exhibit finite delays and are modeled with delay elements while analyzing their behavior. A delay element is said to have pure delay if it delays the transition of its input waveform to its output but does not otherwise alter the waveform itself. On the other hand, a delay element is said to exhibit inertial delay with threshold d_i if it does not react to input pulses shorter than d_i and the pulses longer than d_i are transmitted to its output with a delay d_i. A delay element is said to have bounded delay if its delay can take any value within a given interval and is said to have unbounded delay if its delay can take any finite value. Using this definition, logic circuits can be characterized with a bounded delay model, unbounded gate delay model, or unbounded wire delay model (9). The bounded delay model associates a bounded delay with both logic gates and interconnecting wires. In contrast, the unbounded gate delay model associates an unbounded delay with every gate, and the interconnecting wires are assumed to have zero delay, whereas the unbounded wire delay model associates an unbounded delay with both logic gates and the interconnecting wires. Because real gates and wires have finite delays, output signals may glitch before settling down to their final value. In a synchronous design, clock signal controls all the state changes and communication between vari-

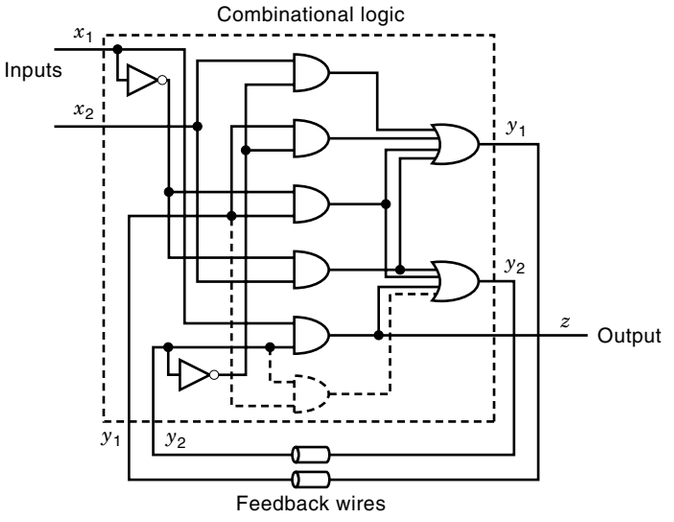

Figure 5. Hazards in asynchronous circuits.

ous components. Between active clock edges, combinational logic generates the next state function, possibly producing many spurious output values (glitches). Synchronous logic operates correctly by ensuring that all the outputs are correct and stable before the next active clock edge. Because asynchronous circuits do not employ a global clock, glitches cannot be filtered out by simply controlling a global synchronization signal. In asynchronous circuits, these glitches are known as hazards, which can cause the circuit to malfunction.

For example, consider the circuit shown in Fig. 5. For this circuit, if input x_1x_2 changes from 00 to 10, then output z goes high, and the next state variable Y_1 changes to 1 while the next state variable Y_2 remains at 1. In this process, the output of the AND gate $\overline{x_1}y_1$ changes to 0 while the output of the AND gate x_1y_2 changes to 1. If the AND gates have nonzero finite delays and one AND gate has more delay than the other, then the output of the OR gate implementing Y_2 can go to 0 for a short duration (equal to the difference in the delays of the AND gates). If this difference in AND gate delays is large enough, the circuit can stabilize in some other state because the transient can be propagated back through the feedback wires. This transient on the output of the OR gate is known as a hazard. Depending on the specifications, changing the logic implementation can eliminate some type of hazards. For example, adding an AND gate y_1y_2 (shown as dotted in Fig. 5) to the circuit implementing Y_2 keeps the output Y_2 at 1, and no hazard can occur assuming that the feedback wires have much larger delays than the delays in combinational logic. If this assumption is not met and y_1 falls too fast, then the output of gate y_1y_2 may fall to give a transient Y_2 output again.

We can now define a hazard as a deviation of an output signal from the specified behavior. A hazard can be classified as a static hazard, a dynamic hazard, a combinational hazard, or a sequential hazard. These are formally defined as follows. A signal is said to have a static hazard if it should remain constant but it changes twice or more (in opposite directions). A static hazard is a 0-hazard if the signal should remain 0, and it is a 1-hazard if the signal should remain 1. A signal is said to have dynamic hazard if it should change only once but it changes multiple times. A combinational hazard occurs as a result of a distribution of finite gate delays in the combinational logic (as described in the preceding example). A combinational hazard is classified as logic hazard if the hazard depends on the particular logic implementation, as described in the preceding example. A logic hazard can be eliminated by changing the logic implementation of the function. A combinational hazard is classified as a function hazard if it cannot be eliminated by changing the logic, irrespective of the gate delays. A sequential hazard occurs as a result of the feedback wire delays. A sequential hazard is called an essential hazard if it is inherent in the finite state machine specifications and occurs irrespective of the logic implementation. For example, the hazards due to critical races that cannot be eliminated by proper state assignment are essential hazards. There is a wide body of literature on hazard analysis and elimination that establishes several properties of static, dynamic, combinational, and sequential hazards. Further details on hazards in asynchronous circuits can be obtained from Unger (15,16).

Given various delay models, asynchronous circuits are generally classified into speed-independent circuits and delay-insensitive circuits. *Speed-independent circuits* (10) operate correctly (without hazards) irrespective of gate delays, and wires are assumed to have zero delays. Thus, their operation is defined using the unbounded gate delay model. In contrast, *delay-insensitive circuits* (11) operate correctly (without hazards) irrespective of the gate as well as wire delays. Thus, their operation is defined using the unbounded wire delay model. In general, digital logic circuits are designed with basic logic gates such as a NAND gate and NOR gate. It is known that to ensure delay-insensitive behavior of the circuit, any logic gate must wait for a transition on all of its inputs before generating a transition on its output (12). Because all single-output standard logic gates such as AND, NAND, OR, NOR do not satisfy this constraint, they cannot be used to build delay-insensitive circuits. Thus, only logic elements that satisfy this constraint, such as C-element (which implements an AND of transitions), inverters, buffers, and wires, can be used to implement delay-insensitive circuits. In order to design delay-insensitive asynchronous circuits in practice, more complex delay-insensitive logic elements with a range of functionality are used (13). Because it is impractical to design pure delay-insensitive circuits with simple logic elements, researchers have relaxed the pure delay-insensitivity constraint to develop quasi-delay-insensitive circuits. Quasi-delay-insensitive circuits are similar to delay-insensitive circuits except that they assume isochronic forks (72). An isochronic fork is a forked wire where all branches have exactly the same delay. Some researchers have relaxed this constraint to allow a bounded skew between different branches of the fork. In contrast, the delays on different branches of a fork in delay-insensitive circuits are completely independent of each other. In additon to speed-independent and delay-insensitive circuits, timed asynchronous circuits (14,45) have also been developed which utilize the bounded nature of delays in practice to optimize the performance of asynchronous implementation.

There are several approaches of designing asynchronous circuits. Asynchronous design methodologies can be characterized in many ways (17,18) (i.e., the delay model used for their implementation, the type of design specifications used to specify their behavior). In the following section, several major design methods for implementing asynchronous designs are

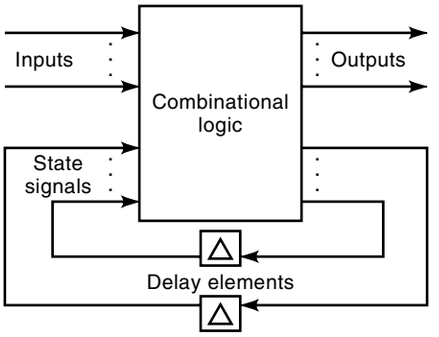

Figure 6. Asynchronous Huffman state machine.

briefly described. These methods are categorized according to the type of specifications they use.

ASYNCHRONOUS DESIGN METHODOLOGIES

Asynchronous Design Using Finite State Machines

Huffman State Machines. Asynchronous behavior is traditionally synthesized using finite state machine specifications, which are implemented using a Huffman machine (15) shown in Fig. 6. A Huffman machine follows a bounded delay model and consists of combinational logic with primary inputs, primary outputs, and feedback state variables. Instead of latches or flip-flops in the state signal feedback loop, delay elements are used to store state information. In each state, a state machine can receive inputs, generate outputs, and move to the next state. If the state machine receives an input and remains in the same state, then that state is called a stable state. Because asynchronous circuits do not have a clock, state transitions are caused by input changes. If an input change causes a transition from a stable state to an unstable state, then the state continues to change until a stable state is reached. Asynchronous state machines can also be characterized according to the number of inputs that may change between any state transition. In a single-input change machine, only one input may change at any given time, and any subsequent input change takes place only after the circuit has stabilized in response to this input. In contrast, in a multiple-input change-asynchronous state machine, any number of inputs may change simultaneously (i.e., within a very narrow interval d_i) and any subsequent input changes take place only after the circuit has stabilized.

The asynchronous FSM specifications are described using a flow table (15). A flow table expresses a relationship between present states, inputs, next states, and outputs. A flow table represents the output and next-state circuit behavior as a function of its inputs and present state. In Table 1, asynchronous flow table specifications are illustrated with a simple asynchronous modulo 4 counter having two binary inputs x_1 and x_2 and two binary outputs z_1 and z_2. The input $x_1 = 0$, $x_2 = 0$ is a reset input that clears the counter to an initial state producing the output 00. In this example, only a single-input variable is allowed to change in a transition (single-input change mode). The count is incremented by one when the input $x_1 = 0$, $x_2 = 1$ is received and incremented by two when the input $x_1 = 1$, $x_2 = 0$ is received. The count remains the same for input $x_1 = 1$, $x_2 = 1$. The outputs z_1 and z_2 represent the counter outputs (i.e., the count). Table 1 shows the flow table specifications for the asynchronous counter. For each state S_i of the flow table, next, state entries are specified for different input signals, and stable states are denoted by a box. The rest state a is stable in input column 00. This input may be followed by input 01, which results in a next state b and outputs $z_1 = 0$, $z_2 = 1$ (i.e., a count of 1). Similarly, input 10 after the input 00 results in a next state c and outputs $z_1 = 1$, $z_2 = 0$ (i.e., a count of 2). The next-state entry and the outputs in column input 10 for present state a is unspecified. The stable b in input column 01 may be followed by input 00 or 11, taking the state machine to reset state a or state d, respectively. In this case, the counter outputs z_1 and z_2 remain unchanged (i.e., the count remains the same). Similarly, the rest of the flow table entries can also be specified, as shown in Table 1. Outputs are specified only for stable states, assuming that outputs may change at any time during the transition.

The initial flow table describing the desired asynchronous behavior usually contains some redundancy. The removal of redundant states [i.e., state minimization] is important to reduce the circuit complexity of the final implementation. To implement the reduced state table as a logic circuit, every state must be assigned a unique Boolean encoding. This step is known as state assignment (15). In the case of a transition between states that involve multiple state signal changes, it must be ensured that the circuit operation is independent of the order in which these state signals change. A situation in which more than one state variable must change in the course of a transition is called a race condition. A race is said to be a critical race if the stable state reached by the circuit depends on the order in which the signals change (i.e., the outcome of the race). There are many techniques to achieve critical race free state assignment such as one-hot assignments (15) or those described by Tracey (19). Further details on various state minimization and state assignment techniques can be obtained from Unger (15).

The minimized and critical race free encoded flow table describes a set of logic equations that produce the next state and output functions. These logic functions can be implemented by a combinational circuit. The logic function of a next state (i.e., feedback wires) or an output signal can be obtained by deriving its Karnaugh map (i.e., finding its implied value in each binary encoded flow table state). Connecting the next-state output and present-state input of this circuit yields the logic circuit implementing the asynchronous state machine. This implementation may exhibit a hazardous behavior. Hazard removal is much easier in single-input change machines as compared to multiple-input change machines because of their more constrained operation as discussed previously. The issue of hazard removal can be simplified if it is assumed that the inputs to the logic circuit do not change until the circuit outputs have stabilized. This condition is known as *fundamental mode constraint* which generally results in severe performance penalty. Fundamental mode constraint is similar to a hold time constraint for a simple latch or a flipflop. Several formal design procedures exist for synthesizing hazard-free fundamental-mode single-input change asynchronous state machines and are described in (15). Although multiple-input change state machines are more flexible, they are significantly more difficult to design.

Table 1. An Example Flow Table Specifications

Present State	$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 11$	$x_1x_2 = 10$
a	$\boxed{a}$/00	b/−	−/−	c/−
b	a/−	$\boxed{b}$/01	d/−	−/−
c	a/−	−/−	e/−	$\boxed{c}$/10
d	−/−	f/−	$\boxed{d}$/01	g/−
e	−/−	h/−	$\boxed{c}$/10	i/−
f	a/−	$\boxed{f}$/10	e/−	−/−
g	a/−	−/−	j/−	$\boxed{g}$/11
h	a/−	$\boxed{h}$/11	j/−	−/−
i	a/−	−/−	k/−	$\boxed{i}$/00
j	−/−	m/−	$\boxed{j}$/11	l/−
k	−/−	b/−	$\boxed{k}$/00	c/−
l	a/−	−/−	d/−	$\boxed{l}$/01
m	a/−	$\boxed{m}$/00	k/−	−/−

Some methods (15) for synthesizing multiple-input change machines rely on inertial delays to filter out glitches. Unfortunately, inertial delays are difficult to build, and they penalize the logic implementation in terms of performance. In the absence of fundamental mode constraint, it is extremely difficult to obtain hazard-free behavior in Huffman state machine implementations. In addition, to obtain a hazard-free implementation, the Huffman state machine specifications must be free from essential hazards (i.e., hazards inherent to the specification). Burst mode specifications developed by Nowick et al. (22) provided solutions to these hazard-free implementation problems. In addition to providing the flexibility of multiple-input changes, burst mode specifications avoid essential hazards by construction.

Self-synchronized State Machines. The difficulties and overhead of hazard elimination associated with Huffman state machines gave rise to an asynchronous design style called self-synchronized state machines (20,21). As shown in Fig. 7, these state machines generate their own synchronization signal, which acts like a clock on internal flip-flops. The self-synchronized machine consists of combinational logic, storage elements with clock control, inputs and outputs, and state variables that are fed back to machine inputs. The local clock is generated from the external inputs and the current state.

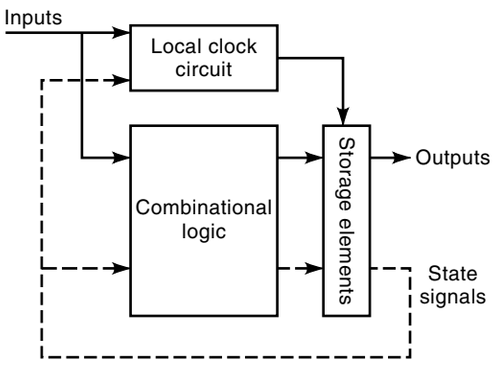

Figure 7. Self-synchronized state machine.

The local clock is also used to eliminate a number of possible hazards.

In general, a self-synchronized machine is idle until an input change occurs. Subsequently, the combinational logic generates corresponding outputs and state signals. The storage elements update the machine state when the local clock generation circuitry generates a clock pulse. The machine is ready to accept new inputs after the storage elements have been updated. Self-synchronized machines try to combine the benefits of synchronous and asynchronous state machines. Both single-input change and multiple-input change self-synchronized machines can be designed. They offer the advantages of synchronous machines (i.e., they are simple and do not require critical-race free state assignments). In addition, they do not require hazard-free logic for outputs and next-state entries. However, they do transfer the problem of hazard elimination to the local clock generation logic, which may require an inertial delay at its output to eliminate hazards. In addition, they may have poor performance as a result of their design based on the worst-case delay of output and state variables.

Burst Mode State Machines. Nowick et al. (22) made significant contributions to the design of asynchronous circuits using state machines by developing burst mode state machine specifications. One of the biggest advantage of burst mode machines is that the logic implementation is guaranteed to be hazard-free, while maintaining high performance. In addition, burst mode machines allow multiple-input changes, thereby yielding more concurrent systems. In burst mode state machines, a set of input transitions (i.e., input burst) is followed by a set of output transitions (i.e., output burst). Finally the state change takes place. The inputs in the input burst may occur in any order. The output burst can occur only after the entire input burst has occurred. A new input burst cannot occur until the machine has reacted to the previous input burst. Thus, these specifications also require the fundamental mode assumption but only between transitions in different input bursts. In order to distinguish different input bursts in a given state, no input burst can be a subset of another input burst. This constraint is known as maximal set constraint. In addition, each state must follow the unique en-

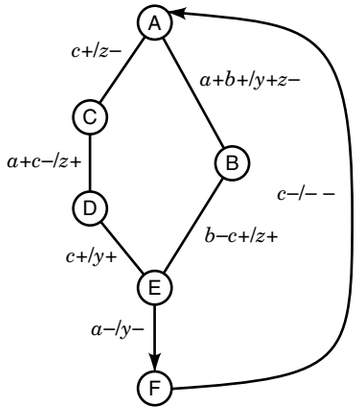

Figure 8. Burst mode specifications of a simple controller.

try constraint, which implies that every state has a unique entry point. Thus, a given state is always entered with the same set of input values. Burst mode specifications are specified using state diagrams. Figure 8 shows the burst mode specifications of a simple controller with three inputs a, b, c and two outputs y, z. Each transition between two states is labeled with an input burst followed by an output burst. For example, transition between state A and state B takes place with an input burst of $\{a+\ b+\}$ and generates an output burst of $\{y+\ z-\}$, where $a+$ denotes a rising transition on input a and $z-$ denotes a falling transition on output z. The key difference between burst mode state machines and multiple-input change Huffman state machines is that, unlike multiple-input change machines, inputs within a burst could be uncorrelated, arriving in any order and at any time.

Burst mode asynchronous specifications are implemented using self-synchronized or locally clocked state machine shown in Fig. 7. Implementation of burst mode specification in locally clocked machines differs from self-synchronized machines in several aspects. In burst mode implementation, the clock is generated selectively (i.e., some transitions do not require a clock pulse). In addition, unlike many self-synchronized methods (20,21), the clock unit does not require inertial delays to eliminate hazards. As mentioned previously, burst mode specifications impose simple constraints on input transitions, such as maximal set and unique entry constraint to guarantee hazard-free logic. Initially, the burst mode locally clocked machine is stable in some state. Inputs in a specified burst may then change value in any order and at any time. Throughout this input burst, the machine outputs and state remain unchanged. When the input burst is complete, the outputs change as specified. A state change may also occur concurrently with the output change. Then the machine will be driven to a new stable state. It is also possible for the input burst and output burst to occur without a state change. In either case, no further inputs may arrive until the machine is stable. After the machine is stable, the transition is complete and the machine is ready to receive a new input burst. Throughout the entire machine cycle, outputs and state variables must be free of glitches.

Although this design style allows multiple input changes that can arrive at arbitrary times, it is restricted in terms of modeling concurrency between input and output signals. Yun et al. (23) removed some of these restrictions with extended burst mode specifications. To implement large designs, usually the behavior is described using distributed specifications. Kudva et al. developed an approach for synthesizing distributed burst mode specifications (24). Nowick et al. have developed a comprehensive suite of burst mode synthesis tools which include hazard-free logic minimization capability (24a) as well.

Burst mode specifications can also be implemented using the Huffman-style state machines illustrated in Fig. 6. Yun and Dill (25) proposed automatic synthesis techniques for burst mode specifications using Huffman state machines, also known as 3D asynchronous state machines. Burst mode specifications have enjoyed significant success in industry as well. Research by Davis et al. (26) at Hewlett-Packard Labs resulted in a complete CAD methodology for burst mode synthesis, which was used to develop several industrial designs.

Asynchronous Design Using Petri Nets-Based Specifications

Petri nets (27) are a modeling tool for the study of systems. A petri net is a bipartite directed graph $\langle P,T,F,M_0\rangle$, consisting of a finite set of transitions T, a finite set of places P, and a flow relation $F \subseteq P \times T \cup T \times P$ specifying a binary relation between transitions (represented as bars) and places (represented as circles). For example, in the petri net of Fig. 9, six places p_0, p_1, p_2, p_3, p_4, and p_5 correspond to six conditions (i.e., a job to be processed, a job is waiting, the processor is idle, a job is being processed, a job is waiting to be output, and a job has been processed). This example petri net has four transitions t_1, t_2, t_3, and t_4 that correspond to four events (i.e., a job is put in input queue, a job is started, a job is completed, and a job is output).

The net structure represents the static nature of the modeled system. Its dynamic behavior is captured by its markings and the firing of transitions, which transform one marking into another. A marking M is a collection of places corresponding to the local conditions that hold at a particular moment. It is graphically represented by solid circles called tokens, residing in these places, i.e., for a given place p, a marking defines a nonnegative integer representing number of tokens in p. The initial marking is denoted as M_0. A transi-

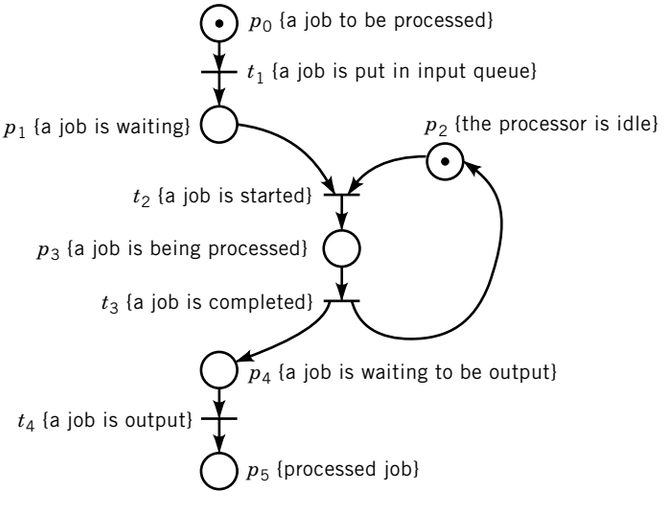

Figure 9. Modeling of a simple computer system with petri nets.

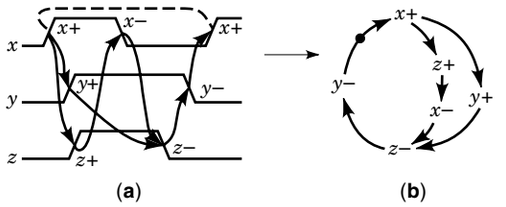

Figure 10. (a) An example timing diagram and (b) the corresponding signal transition graph.

tion is said to be enabled when all its input places are marked with at least one token. The firing of an enabled transition removes one token from each input place and deposits one token in each output place. The transformation of a marking M into another marking M', by firing a transition t, is denoted by $M \xrightarrow{t} M'$. For the petri net example of Fig. 9, the transition t_1 (a job is put in the input queue) is enabled because place p_0 has one token (i.e., condition a job to be processed is valid) and p_0 is the only input place for transition t_1. After the transition t_1 occurs (fires), a token will be removed from place p_0 and will be deposited in transition t_1's output place p_1. This implies that the condition a job is waiting will become valid. The initial marking M_0 of the example petri net (Fig. 9) is $\{p_0, p_2\}$. Occurrence (firing) of transition t_1 transforms the petri net marking to $\{p_1, p_2\}$.

One of the main features of petri nets is the inherent concurrency. In the petri net model, two events, which are both enabled and do not interact, may occur independently. Another major characteristic of petri nets is their asynchronous nature. Petri nets define a partial order of the occurrence of events. The petri net structure contains all the necessary information to define all possible sequences of events. Thus, for the petri net of Fig. 9, the event a job is completed (transition t_3) must follow the event a job is started (transition t_2). Petri net execution is viewed as a sequence of discrete events. The order in which the events occur is one of possibly many allowed by the petri net structure. In addition to representing concurrency, petri nets can also represent nondeterminism, i.e., a *choice* between several events. If, at any time, more than one interacting transitions (i.e., transitions that have a common place as input) are enabled simultaneously, then any of the enabled transitions may fire next. The choice as to which transition fires is made in a nondeterministic manner, i.e., randomly.

Signal Transition Graphs. Signal transition graphs (STG) are event-based graphical models that specify the asynchronous circuit behavior using temporal relationships between signal transitions rather than states. They were introduced independently by Rosenblum and Yakovlev (28) and Chu (29). Signal transition graph specifications are based on petri nets as an underlying formalism, where transitions T of the net are interpreted as rising (positive) and falling (negative) transitions on input I and output O wires of the asynchronous circuit (i.e., $T \in I \times \{+, -\} \cup O \times \{+, -\}$). In an STG, transitions are represented by their names instead of a bar and a label. Every place with a single input and output transition is represented by an arc between these transitions, which represents their temporal relationships [as illustrated in Fig. 10(b)].

STG specifications can explicitly describe asynchronous circuit behavior (e.g., concurrency, causality and conflict) and have captured wide attention. Figure 10(a) shows a timing diagram that specifies three signals x, y, and z. A positive transition of signal y follows the positive transition on signal x. Similarly, a positive transition of signal z follows the positive transition on signal x. Because there is no ordering constraint between the positive transitions of y and z, they are said to be concurrent. This timing diagram can be directly transformed into STG specifications by representing signal transitions as nodes and ordering constraints as directed arcs. The STG specifications corresponding to the timing diagram of Fig. 10(a) are shown in Fig. 10(b).

A model similar to STG specifications, called change diagrams, was proposed by Varshavsky et al. (30). Change diagrams can specify concurrent behavior but are unable to specify conflict behavior (i.e., either one of the events can occur but not both). STG specifications have several advantages for specifying control-intensive asynchronous behavior. STG specifications can explicitly describe the major aspects of asynchronous control circuit behavior (e.g., concurrency, causality, and conflict). Most control-intensive circuits are specified using timing diagrams that can be directly transformed into STG specifications. Thus, they are appealing to designers. Because STG specifications are based on petri nets as an underlying formalism, they can directly use the wide body of petri net analysis techniques. The graphical nature of a signal transition graph makes it easier to analyze circuit behavior at a higher level of abstraction.

A logic circuit can be derived by transforming the STG specifications into a state graph (29) [as shown in Fig. 11(b)]. The state graph represents all the states of STG specifications. It captures all the possible transition sequences in the STG. A state graph can be derived by exhaustively generating all possible markings (i.e., states) of the STG. A state graph can be mapped into a circuit by assigning a unique binary code to each state in the state graph. This binary encoding can be derived from the values of STG signals in each state described as follows.

Logic functions can be derived from a state graph by assigning binary codes to each state. The value of input and output signals in a given state gives the binary coding for that state and is derived using a consistent state assignment defined as follows. *Consistent state assignment:* For STG signals $\{s_1, s_2, \ldots, s_n\}$, a state M in the state graph is assigned a binary code $<M(s_1), M(s_2), \ldots, M(s_n)>$. If a transition t is enabled in state M (i.e., $M \xrightarrow{t} M'$), then $t =$ "s_i+" implies $M(i) = 1$; $t =$ "s_i-" implies $M(i) = 1$ and $M'(i) = 0$.

For example, in the initial marking of STG shown in Fig. 11(a), transition $x+$ is enabled. Thus, signal x must have a value 0 in the initial state because it must go to a value 1 after firing a positive transition of x. Similarly, in the initial state, STG signals y and z can also be evaluated to have values of 0 and 0, respectively. This assigns a binary code 000 to the initial state for state graph shown in Fig. 11(b). The values in binary code 000 correspond to values of STG signals x, y, and z, respectively. Binary encoding for the rest of the states can be easily derived by simply firing the enabled signal transitions and changing the signal values accordingly. Figure 11(b) shows the state graph corresponding to the signal transition graph of Fig. 11(a). Every state the state graph corresponding to the signal transition graph of Fig. 11(a). Ev-

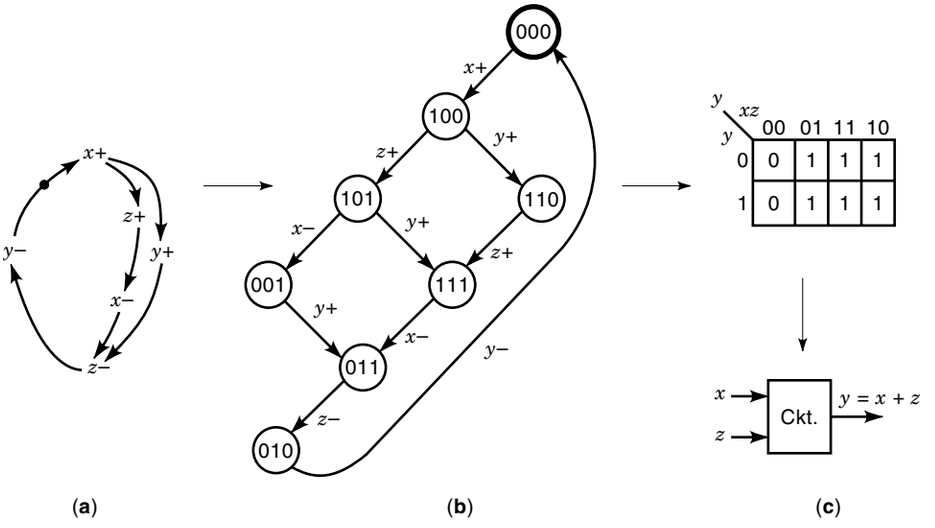

Figure 11. (a) An example STG; (b) the corresponding state graph; and (c) the logic derivation.

ery state in the state graph of Fig. 11(b) has been encoded according to the values of the input and output signals in the STG. The following constraint ensures that the circuit will be able to distinguish between two states by using only the input and output signal values.

A state graph satisfies the complete state coding (CSC) constraint (29) if and only if the transitions of non-input signals, enabled in two states having the same binary code assignment, are the same.

Thus, only input transitions enabled in two states having the same binary code are different, and it is assumed that the environment can distinguish between them. A state graph satisfying the CSC constraints has a well-defined logic function, and there is no conflict of implied values even if the binary code assignments of two states are the same. Because only input and output signal values are used for coding the states, the CSC constraints are necessary for logic implementation of the STG specifications. A CSC violation must be corrected by inserting extra signals in the STG, so as to distinguish between the states violating CSC (31). For example, Fig. 12 shows two states S_1 and S_2 having the same state encoding 011 in a state graph. State S_1 enables an input signal ai transition and an output signal ao transition. On the other hand, state S_2 enables only an output signal bo transition. Because both states S_1 and S_2 have the same binary encoding 011, and they enable different output (i.e., noninput) signals, they violate the CSC constraint according to its definition given previously. This violation can be corrected by inserting another signal n (called a state signal) in the state graph. This signal must have a value of 1 in one state and 0 in the other state. The state signal insertion to satisfy complete state coding violation is also illustrated in Fig. 12 where n is assigned to the binary code of state S_1 and $\overline{n}$ is assigned to the binary code of state S_2. Thus, the states S_1 and S_2 now have different binary codes (i.e., 001n and 001$\overline{n}$, respectively).

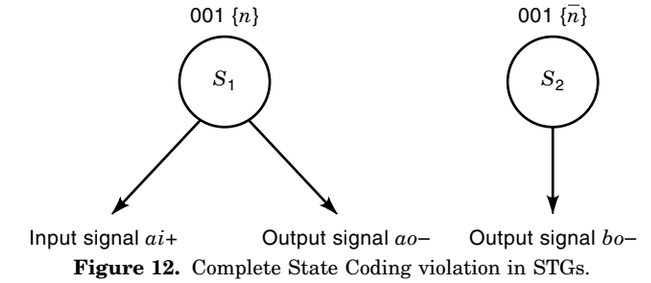

Figure 12. Complete State Coding violation in STGs.

A logic circuit can be derived from a state graph with no CSC violations by finding the implied values of output signals in every state. The implied value of an output in a state graph state is obtained as follows. The implied value of an output o in some state graph state s is defined as

- the complement of the present value of o in binary code of state s, if o is enabled in state s, or
- the present value of o in the binary code of state s, if o is not enabled in state s.

For example, in initial state 000 of state graph in Fig. 11(b), output y is not enabled. Thus, the implied value of y in state 000 will be same as the present value of y (i.e., 0). On the other hand, in state 100, output transition $y+$ is enabled. Thus, the implied value of y in state 100 will be the complement of the present value of y (i.e., 1). Similarly, the implied value of output y in the rest of the state graph states can also be obtained. The logic function of an output can be obtained by constructing a Karnaugh map that contains the entries of output-implied values. For example, in the Karnaugh map of output y in Fig. 11(c), the entry corresponding to $xyz = 000$ is 0 (i.e., the implied value of y in state graph state 000) and the entry corresponding to $xyz = 100$ is 1 (i.e., the implied value of y in state graph state 100). A Karnaugh map gives the logic function for the corresponding output [e.g., the Karnaugh map in Fig. 11(c) generates the logic function $y = x + z$].

It has been proved that CSC is the necessary and sufficient requirement for implementing hazard-free STG specifications into logic circuits, if the logic can be implemented with a single complex gate (32). A number of state encoding techniques have been developed to satisfy CSC constraint. Lin and Lin (37) and Vanbekbergen et al. (38) proposed state encoding techniques to satisfy a more strict constraint than the general CSC constraint. This constraint requires that each state in

the state graph must have a unique state code. In addition, these techniques were restricted to STG specifications describing only concurrent asynchronous behavior (also known as marked graphs). These techniques were also limited by an additional restriction that for every signal, the STG can specify only one rising and one falling transition. Lavagno et al. (31) proposed a new state encoding framework for STG specifications with a limited interplay of concurrency and choice. They solved the CSC constraint satisfaction problem at the state graph level by transforming the STG into an FSM state table. This state table is then reduced with state minimization and encoded using critical race-free state assignment techniques. This approach guarantees sufficient conditions for the CSC satisfaction. The algorithm inserts state signals into the original STG to satisfy CSC constraints, but only handles live free-choice petri nets. The CSC solutions obtained in this framework correspond to a special class of STG transformations. Vanbekbergen et al. (40) proposed a general framework to solve the CSC satisfaction problem for general STG specifications. It is not limited to marked graph or free-choice petri nets. They formulated the CSC problem as a Boolean satisfiability (SAT) problem. They gave the necessary and sufficient conditions for the insertion of state signals. This ensures that CSC property while conserving the original STG behavior. It is well known that many combinatorial optimization problems can be directly transformed into the SAT problem. Unfortunately, the instances of SAT formulas derived from practical STGs are too large to be solved efficiently. Puri and Gu (41) proposed an efficient modular approach for solving complete state coding problems by first partitioning the signal transition graph into a number of simpler and manageable smaller modular graphs. This approach is applicable to general signal transition graphs and achieves significant performance improvement (42,43). A group of researchers that include Cortadella, Kishinevsky, Kondratyev, Lavagno, Pastor, Semenov, and Yakovlev et al. (43a,43b,43c) have made significant progress in reducing the complexity of solving complete state coding. They solved the CSC problem by directly analyzing the STG specifications rather than state graphs derived from this STG. Because the number of states in a state graph can be very large for a highly concurrent STG, methods that work directly at the STG level can yield significant performance improvement in some cases.

In general, logic circuits derived from practical STG specifications are too large to be implemented in a single complex gate. This implies that the gate level logic implementation obtained from state graphs satisfying CSC constraint may not be free from all hazards (32). Lavagno et al. (34) and Yu and Subrahmanyan (39) developed heuristic techniques that add delays for gate-level hazard removal for bounded delay models. Moon proved in (33) that an STG that satisfies the CSC requirement is free from all functional hazards, all critical races, and all static O-hazards, under the unbounded gate delay (speed-independent) model. Thus, the speed-independent logic implementation of an STG-satisfying CSC constraint may have only static 1-hazards and dynamic hazards. Moon et al. further proposed algorithms to remove these remaining hazards (32,33). Significant progress was made by Kondratyev et al. (43d), who developed sufficient conditions called monotonous cover condition and unique entry condition to derive a hazard-free speed-independent (i.e., under unbounded gate delay) circuit. Pastor and Cortadella (43e) developed efficient algorithms for hazard-free synthesis of speed-independent circuits directly from STGs that satisfy CSC constraint. Beerel and Meng (35) and Kishinevsky et al. (36) also developed efficient algorithms to derive hazard-free speed-independent circuits from state graphs. Myers and Meng (45) have extended the STG specifications to incorporate timing constraints and developed efficient synthesis algorithms to implement them.

Over several years, a group of researchers, which include Kishinevsky, Kondratyev, Taubin, and Varshavsky, have made significant contributions to developing a publicly available design tool called FORCAGE for synthesizing practical STG specification into speed-independent circuits (36). Another publicly available design tool for synthesizing hazard-free circuits from STG specifications was developed by Lavagno et al., and this tool is integrated with the publicly available Berkeley SIS tool (45a). Vanbekbergen, Ykman, and Lin et al. at IMEC Belgium developed a tool called ASSASSIN for synthesis and analysis of asynchronous control circuits from general STGs including timed signal transition graphs (45b). Cortadella et al. have developed a state encoding and synthesis tool called PATRIFY for designing speed-independent circuits from STG specifications (43b,43c).

Asynchronous Design Using Communicating Process

CSP-Based Specifications. Hoare introduced a specification language, called communicating sequential processes (CSP) (46), for a set of concurrent processes that communicate on fixed links called channels. Martin used a subset of this specification language consisting of sequential, communication, and probe constructs and developed a rule-based asynchronous synthesis procedure that transforms CSP (communicating sequential processes) specifications describing asynchronous behavior into CMOS (complimentary metal oxide silicon) circuits (47,48).

In Martin's method, asynchronous behavior specified using the CSP notation is transformed into a semantically equivalent set of VLSI (Very Large Scale Integrated circuit) operators using transformations such as process decomposition, handshaking expansion, and production-rule expansion. The first step of the transformation, called process decomposition replaces one process with several processes by application of a decomposition rule. Process decomposition makes it possible to reduce a process with an arbitrary control structure to a set of subprocesses of only two different types: either a (finite or infinite) sequence of communication actions or a repetition of process selections. The next step of the transformation, called handshaking expansion, replaces each channel with a pair of wire-operators and each communication action in a program with its implementation in terms of elementary actions of four-phase handshaking protocol. Production-rule expansion is the transformation from a handshaking expansion to a set of production rules. It is the most important step of the compilation and consists of state assignment, guard strengthening, and symmetrization. State assignment transforms the handshaking expansion to ensure that each state of the expansion is unique. After state variables have been introduced so as to distinguish any two states of the hand-

shaking expansion, it is possible to strengthen the Boolean guards of the production rules to enforce program-order execution. Subsequently, symmetrization may be performed on production rules to minimize the number of state-holding operators. Finally, the production rules can be transformed into a circuit implementation.

Burns developed an automated version of this procedure and further improved the performance of implemented circuits (49,50). This design style assumes a four-phase handshaking protocol [Fig. 2(b)]. The designed circuits are quasi-delay-insensitive (i.e., the operation of the circuit is independent of the delays of the component and interconnecting wire delays) and the delays of wire forks are comparable [i.e., the design conforms to the isochronic fork assumption (51)]. This method using the CSP specifications has many practical examples such as the distributed mutual exclusion element (52) and an asynchronous microprocessor (53). A similar compilation-based method that generalizes Martin's communication style to include shared variables has been proposed by van Berkel et al. (54). These techniques have also been implemented by van Berkel et al. into a robust asynchronous design tool called TANGRAM at Philips Research Labs. TANGRAM first compiles the CSP-based specifications into an intermediate representation called handshake circuit. A handshake circuit consists of a network of handshake processes which communicate asynchronously on channels using asynchronous protocols. The circuit is then optimized using peephole optimization, and finally the components are mapped to VLSI implementation. TANGRAM has been successfully used to implement several DSP (Digital Signal Processing) designs at Philips. Brunvand and Sproull used a programming language called occam, which is based on CSP (55). Occam describes asynchronous computations as a set of concurrent processes that interact by communication over channels. In occam, control over concurrent and sequential aspects of communication is explicit. Brunvand and Sproull developed a design methodology to translate programs written in a subset of occam automatically into delay-insensitive circuits using syntax-driven techniques and two-phase handshaking protocol [Fig. 2(a)]. The resulting circuits are then improved using semantics-preserving circuit-to-circuit transformations.

Trace Theory-based Specifications. Trace theory was inspired by Hoare's CSP and developed by van de Snepscheut (56) and Rem et al. (57). Ebergen used trace theory to described asynchronous behavior (58). A trace of a circuit represents a history of execution by listing all the transitions of signals at its interface. The set of all possible traces of a circuit, known as trace set, completely specifies the behavior of the circuit at its interface. More formally, a trace structure is defined using the communication alphabet Σ of a circuit. This alphabet consists of a finite number of symbols used to represent the wires over which a circuit communicates. A trace structure is defined as a triple: $T = \langle I,O,X \rangle$, where $I \in \Sigma$ is a finite set of input symbols, $O \in \Sigma$ is a finite set of output symbols, and $X \in \Sigma^*$ is a set of all possible traces of the circuit. The set Σ^* is the set of all finite-length sequences of symbols in Σ. Although an individual trace represents a single execution history of the circuit, the trace set X captures all possible execution histories of an interface and thus completely specifies the behavior of the interface. A trace containing no execution symbols is represented by symbol ϵ. Traces may be extended by appending a new symbol corresponding to a possible transition onto the end of the trace. This indicates that the transition is allowed to occur immediately following the events already recorded in the trace. Trace sets, being sets of simple lists of symbols, are often expressed using familiar regular-expression notation. This notation makes the description of an entire trace set more compact than simply listing all the possible traces. Regular expressions are composed of the symbols in the alphabet $\Sigma \cup \{\epsilon\}$ of the trace structure, and special symbols $|$, $*$, (,) interpreted as follows.

Let r_1 and r_2 be two regular expressions.

- r_1r_2 is a regular expression representing the concatenation of r_1 and r_2.
- $r_1|r_2$ is a regular expression representing a choice between expressions r_1 and r_2.
- r_1^* is a regular expression that represents zero or more repetitions of expression r_1 into a single expression.
- (r_1) represents a regular expression that groups all the symbols of expression r_1 into a single expression.

For example, if alphabet $\Sigma = \{r,a\}$ is used to represent request-and-acknowledge wires in a two-phase signaling protocol, the trace set of this protocol can be expressed with the regular expression $\{[(ra)^*][(ra)^*r\}$.

In designing delay-insensitive asynchronous circuits through trace theory, components are described using commands that describe sequences of possible events (i.e., traces). Based on trace theory, Ebergen developed the concepts of formal decomposition of a component (59). A decomposition of a component represents a realization of that component by means of a network of other basic components such that the correctness of the network is insensitive to delays in the basic components. The basic components in trace theory are implemented with C-elements, a XOR gate, Toggle elements, and Merge elements (59). All communication in this method is through a two-phase handshaking protocol.

Asynchronous Design Using Micropipelines

Pipelines provide an efficient framework for performing high-speed computations because their separate stages can operate in parallel. Pipelines both store and process data, and the storage elements (registers) and processing logic blocks alternate along the length of the pipeline. Thus, without any processing logic blocks, a pipeline will act like a shift register. Synchronous circuits use clocked pipelines [as shown in Fig. 13(a)], where data advance through the pipeline at fixed clock rate. Because processing logic blocks in the pipelines may have different delays, the clock rate is chosen according to the worst-case delay of any processing block. Because of this fixed clock rate, a clocked pipeline operates at a much slower data rate than its optimal performance. This drawback of clocked pipelines can be eliminated by employing asynchronous pipelines where different stages operate at different rates and they communicate with each other using handshaking protocols. Micropipelines were introduced by Sutherland (8) as an asynchronous alternative to synchronous pipelines. As shown

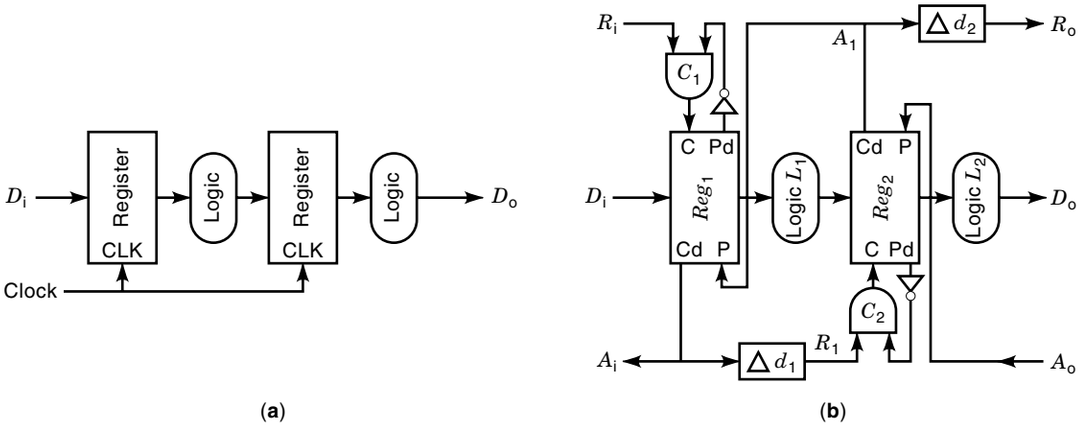

Figure 13. (a) Clocked pipeline and (b) micropipeline with processing.

in Fig. 13(b), a micropipeline consists of alternating logic blocks separated by registers and control circuitry. Computation on data in a micropipeline is accomplished by the logic blocks. In addition to its elastic nature, a major benefit of micropipeline structure is that hazards can be filtered out by the registers that moderate the flow of data through the logic blocks.

In micropipelines, event-controlled registers are employed as opposed to clocked registers in clocked pipelines. An event-controlled register responds to events on its two control wires, called capture [denoted by C in Fig. 13(b)] and pass [denoted by P in Fig. 13(b)]. When the control signals C and P are in same state, the event-controlled register is transparent (i.e., its data input is directly connected to its output). When signals C and P differ in their state, the event-controlled register captures and stores data, and its data output is disconnected from its data output. The behavior of this event-controlled register can also be described in terms of events. Initially, the register is assumed to be transparent. In addition, it is assumed that the capture control signal and the pass control signal always alternate. An event on the capture control wire causes the register to capture and hold the data value passing through it. This event isolates the output value of the elements from changes at the register's input but does not change the output value. A subsequent event on the pass control wire returns the register to its transparent state, permitting the next data value to appear at its output and possibly changing its output value. Thus, after each event on the pass control wire, a new output value may appear. The event-controlled register also includes two control outputs called capture done (denoted as Cd) and pass done (denoted as Pd), which are amplified and thus delayed versions of the corresponding control input signals capture and pass. The control output signals Cd and Pd deliver output events after the register has done its action (i.e., they must be delayed until after the register has performed the corresponding capture and pass actions).

The datapath of a micropipeline as shown in Fig. 13(b) consists of event-controlled registers and logic blocks for data processing, which alternate along the length of the micropipeline. The control for the micropipeline consists of a string of Muller C-elements with inverters interposed as illustrated in Fig. 13(b). Initially, the outputs of all Muller C-elements are initialized to the same state. This is achieved through a master clear signal. A request event received on the R_i wire of the micropipeline initiates the processing. This allows C-element C_1 to generate a capture signal for register Reg_1. Subsequently after some delay, register Reg_1 generates a capture done (Cd) signal, which sends an acknowledge event back on A_i wire. The event on Cd wire generated by register Reg_1 also reaches C-element C_2 after being delayed by a duration more than the worst-case delay d_1 of the logic block L_1, which generates a capture signal for register R_2. This indicates that logic block L_1 has completed processing and register R_2 should capture this data. After register Reg_2 captures data, it generates a capture done signal that in turn signals register Reg_1 through pass (P) signal that it can pass new data to logic block L_1 now. At this point, logic block L_2 can process the captured data in register Reg_2 concurrently with processing in logic block L_1. After the event on its pass wire, register Reg_1 generates an event on its pass done (Pd) signal to indicate that it has switched to the transparent mode. With this event on Pd signal of register Reg_1, C-element C_1 will again generate a capture signal for register Reg_1 after a request event on R_i is received signaling the availability of data and the same process continues.

Micropipelines have been extensively used in one of the most comprehensive research efforts to design an asynchronous microprocessor called AMULET (60), an asynchronous version of the popular synchronous ARM microprocessor. Although micropipelines avoid the hazard problem and are elastic in nature (i.e., each pipeline stage can have different delay), they still deliver worst-case performance in each single state of the micropipeline.

ASYNCHRONOUS PROCESSORS AND OTHER DESIGN EXAMPLES

Early digital computers embodied a variety of design styles. Although most designs were based on synchronous techniques, there were several examples that used an asynchronous approach. For example, ORDVAC, built at the University of Illinois in 1951 and IAS, built by John Neumann's group at Princeton University in 1951, were asynchronous designs and operated without any central clock. Later in 1974,

design team at Manchester University built a processor called MU5 that used asynchronous control. In 1978, Davis designed the first dataflow computer, called DDM-1 (61), which used locally synchronous modules that communicated using request-and-acknowledge protocol. While modern digital computers are dominated by synchronous approach, some asynchronous processor designs are beginning to make the transition from research to products.

Alain Martin at Caltech developed the first quasi-delay-insensitive 16-bit asynchronous microprocessor (53) that was fabricated in 1.6 μm CMOS version and consumed only 200 mW at 5 V and 7.6 mW at 2 V. This design was later implemented using GaAs technology as well (62). Recently Martin et al. also designed asynchronous version of MIPS R3000 processor (63). Researchers at Manchester University have designed several asynchronous versions of the ARM microprocessor, called AMULET (60). The most recent version of this asynchronous microprocessor, known as AMULET2e (64), delivers 40 MIPS at only 150 mW and targets portable embedded applications. Advanced RISC (Reduced Instruction Set Computers) Machine Ltd., the inventors of ARM microprocessor, have already initiated the efforts to transfer this asynchronous design into a commercial offering (65). Sharp Corporation recently announced an asynchronous media processor chip called NMP (66), that employs a data-driven architecture. It uses eight clock-free CPUs (Central Processing Unit), each of which delivers 600 MOPS using only 40,000 gates and consuming less than 60 mW. In a significant development in the area of commercial asynchronous processors, Cogency Technology, Inc., revealed a design system (67) that can produce made-to-order, self-timed CPUs, and DSPs. In its first commercial use, the suite has produced a completely asynchronous digital signal-processing chip 'stDSP', which was designed for LG Semicon. This chip is roughly the same size as a functionally identical synchronous version but uses 47% less power. Researchers at Tokyo Institute of Technology recently designed a 32-bit delay-insensitive asynchronous version of a MIPS R2000 processor, called TITAC-2 (68). Similar to any asynchronous processor, TITAC-2 works correctly even with large temperature and power supply variations. In addition, TITAC-2 achieves a performance of 52 MIPS with a power consumption of 2 W at 3.3 V. Researchers at Sun Microsystems have recently designed an asynchronous counterflow pipeline processor (69). A significant design effort by a team of designers at Hewlett-Packard Labs resulted in a completely asynchronous full-custom CMOS chip called Post Office (70), which had 300,000 transistors and was designed to support internode communication for the Mayfly parallel processing system. Mark Dean at Stanford University built a processor called STRiP (self-timed RISC processor) (71), which includes both synchronous and asynchronous design techniques. In addition to the large asynchronous chips mentioned, several smaller asynchronous chips such as a DCC error corrector chip (72) and a high-speed packet switching component chip (73) by Philips Research Labs and a communication chip at Hewlett-Packard Labs (7) are also slowly beginning to make their way into the commercial marketplace.

In spite of all the asynchronous design applications mentioned earlier, almost all commercial digital designs employ synchronous circuits. The major reason for the overwhelming popularity of synchronous circuits is their ease of design. In addition, asynchronous circuits are prone to hazards that may cause a circuit to malfunction. The hazards are naturally filtered out in a synchronous design by choosing a long enough clock period, which ensures that the circuit is in a stable circuit state before the next input changes take place. Thus, high-performance asynchronous logic circuits that are free of hazards are more difficult to design than their synchronous counterparts. In general, the presence of hazards in asynchronous circuits is a major hindrance in their widespread use. In addition, a clear advantage of asynchronous designs over synchronous designs for large-scale high-performance and low-power circuits still remains to be demonstrated in general.

CONCLUSIONS

Asynchronous logic circuits hold the promise of solving the clock distribution and power dissipation problems in high-performance circuits. This is a major motivating factor in the recent resurgence of interest in applying asynchronous design techniques to processor design as well as low-power applications. Asynchronous logic design should not be viewed as a single alternative to synchronous logic design. More accurately, synchronous design is a special case representing a single design point in a multidimensional asynchronous design space, which varies from totally distributed control to global control with clock. Because of the problems of clocks with frequencies in excess of 1 GHz, it is likely that the next generation of processors will use some of the advantages offered by asynchronous logic by implementing circuits with a suitable combination of synchronous and asynchronous design techniques. Although modern digital computers are dominated by the synchronous approach, asynchronous designs have already started to emerge in commercial products. Recent years have seen a surge in research activities related to asynchronous logic design.

This article focused on the logic design aspect of asynchronous circuits. We only discuss a portion of the vast body of literature available on asynchronous design (a complete bibliography in the field is being maintained at Eindhoven University of Technology, Netherlands and can be obtained from async-bib@win.tue.nl). Specification and design of logic circuits is just one of the steps among several crucial steps in a practical design methodology. Testing and verification of implemented circuits is an integral part of any design flow and consumes a dominant portion of the design cycle of a complex system. Verifying that what you implemented is what you specified is critical for avoiding costly errors late in the design cycle. Formal techniques for verifying circuit implementations are especially critical for asynchronous designs because of the subtlety. Research into these new frontiers of practical formal verification techniques for asynchronous designs is the focus of some recent efforts and will be an interesting area for further study. Although much progress has been made toward developing robust asynchronous design and verification techniques, pushing asynchronous designs over competing synchronous implementations in industry still remains a challenge. Now more than ever, it is crucial to evaluate the advantages of already researched design techniques through practical design implementations and develop methods that can be used in design methodologies being practiced by the industry.

BIBLIOGRAPHY

1. B. Gieseke et al., A 600MHz superscalar RISC microprocessor with out-of-order execution, *Int. Solid State Circuits Symp.,* 1997, pp. 176–177.
2. S. B. Furber, Breaking step: The return of asynchronous logic. *IEE Rev.,* **39** (4): 159–162, 1993.
3. P. Song, Asynchronous design shows promise, *Microprocessor Report,* **11** (13): 1997.
4. S. H. Unger, *The Essence of Logic Circuits,* 2nd ed., New York: IEEE Press, 1997.
5. S. B. Furber, Asynchronous design, in W. Nebel and J. Mermet (eds.), *Proc. Submicron. Electr., Il Ciocco, Italy,* 1996, pp. 461–492.
6. P. E. Gronowski et al., A 433MHz 65b quad-issue RISC microprocessor, in *Int. Solid State Circuits Symp.,* 1996, pp. 222–223.
7. A. Marshall, B. Coates, and P. Siegel, Designing an asynchronous communications chip, *IEEE Design & Test Comput.,* **11** (2): 8–21, 1994.

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

8. I. E. Sutherland, Micropipelines, *Commun. ACM,* **32** (6): 720–738, 1989.

8a. M. Dean, T. Williams, and D. Dill, Efficient self-timing with level-encoded 2-phase dual-rail (LEDR), in C. H. Sequin (ed.), *Advanced Research in VLSI,* Cambridge, MA: MIT Press, 1991, pp. 55–70.

8b. T. Verhoeff, Delay-insensitive codes—an overview. *Distributed Computing,* **3** (1): 1–8, 1988.

9. J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits,* New York: Springer-Verlag, 1995.
10. D. E. Muller and W. S. Bartky, A theory of asynchronous circuits, in *Proc. Int. Symp. Theory of Switching,* Harvard University Press, April 1959, pp. 204–243.
11. J. T. Udding, *Classification and Composition of Delay-Insensitive Circuits,* PhD thesis, Dept. of Math. and Comp. Sci., Eindhoven Univ. of Technology, 1984.
12. J. B. Dennis and S. S. Patil, Speed-independent asynchronous circuits, in *Proc. Hawaii Int. Conf. System Sci.,* 1971, pp. 55–58.
13. J. C. Egergen, A formal approach to designing delay-insensitive circuits, *Distributed Comput.,* **5** (3): 107–119, 1991.
14. D. A. Huffman, The synthesis of sequential switching circuits, in E. F. Moore (ed.), *Sequential Machines: Selected Papers,* New York: Addison-Wesley, 1964.
15. S. H. Unger, *Asynchronous Sequential Switching Circuits,* New York: Wiley-Interscience, 1969.
16. S. H. Unger, Hazards, critical races, and metastability, *IEEE Trans. Comput.,* **44**: 754–768, 1995.
17. S. Hauck, Asynchronous design methodologies: An overview, *Proc. IEEE,* **83**: 69–93, 1995.
18. A. L. Davis and S. M. Nowick, An introduction to asynchronous circuit design, in A. Kent and J. G. Williams (eds.), *The Encyclopedia of Computer Science and Technology,* vol. 38, New York: Marcel Dekker, 1997.
19. J. H. Tracey, Internal state assignments for asynchronous sequential machines, *IEEE Trans. Electron. Comput.,* **EC-15**: 551–560, 1966.
20. A. B. Hayes, Stored state asynchronous sequential circuits, *IEEE Trans. Comput.,* **C-30** (8): 596–600, 1981.
21. C. A. Rey and J. Vaucher, Self-synchronized asynchronous sequential machines, *IEEE Trans. Comput.,* **23**: 1306–1311, 1974.
22. S. M. Nowick and D. L. Dill, Synthesis of asynchronous state machines using a local clock, *Proc. Int. Conf. Comput. Design ICCD,* Los Alamitos, CA: IEEE Computer Society Press, October 1991, pp. 192–197.
23. K. Y. Yun, D. L. Dill, and S. M. Nowick, Practical generalizations of asynchronous state machines, *Proc. Eur. Conf. Design Autom. EDAC,* Los Alamitos, CA: IEEE Computer Society Press, February 1993, pp. 525–530.
24. P. Kudva, G. Gopalakrishnan, and H. Jacobson, A technique for synthesizing distributed burst-mode circuits, *Proc. ACM / IEEE Design Autom. Conf.,* 1996.

24a. S. M. Nowick and D. L. Dill, Exact two-level minimization of hazard-free logic with multiple-input changes, *IEEE Trans. Comput.-Aided Des.,* **14** (8): 986–997, 1995.

25. K. Y. Yun and D. L. Dill, Automatic synthesis of 3D asynchronous state machines, *Proc. Int. Conf. Comput.-Aided Design ICCAD,* Los Alamitos, CA: IEEE Computer Society Press, November 1992, pp. 576–580.
26. A. Davis, B. Coates, and K. Stevens, Automatic synthesis of fast compact asynchronous control circuits, in S. Furber and M. Edwards (eds.), *Asynchronous Design Methodologies,* vol. A-28 IFIP Transactions, Elsevier, 1993, pp. 193–207.
27. T. Murata, Petri nets: Properties, analysis and applications, *Proc. IEEE,* **77**: 541–580, 1989.
28. L. Y. Rosenblum and A. V. Yakovlev, Signal graphs: From self-timed to timed ones, in *Proc. Int. Workshop Timed Petri Nets,* Torino, Italy, Los Alamitos, CA: IEEE Computer Society Press, July 1985, pp. 199–207.
29. T.-A. Chu, *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications,* PhD thesis, MIT Laboratory for Computer Science, MIT, June 1987.
30. V. I. Varshavsky (ed.), *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems,* Dordrecht: Kluwer, 1990.
31. L. Lavagno et al., Solving the state assignment problem for signal transition graphs, *Proc. ACM / IEEE Design Autom. Conf.,* IEEE Computer Society Press, June 1992, pp. 568–572.
32. C. W. Moon, P. R. Stephan, and R. K. Brayton, Synthesis of hazard-free asynchronous circuits from graphical specifications, *Proc. Int. Conf. Comput.-Aided Design ICCAD,* IEEE Computer Society Press, November 1991, pp. 322–325.
33. C. W. Moon and R. K. Brayton, Elimination of dynamic hazards in asynchronous circuits by factoring, *Proc. ACM / IEEE Design Autom. Conf.,* IEEE Computer Society Press, June 1993, pp. 7–13.
34. L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, Algorithms for synthesis of hazard-free asynchronous circuits, *Proc. ACM / IEEE Design Autom. Conf.,* Los Alamitos, CA: IEEE Computer Society Press, pp. 302–308, 1991.
35. P. Beerel and T. H.-Y. Meng, Automatic gate-level synthesis of speed-independent circuits, *Proc. Int. Conf. Comput.-Aided Design ICCAD,* Los Alamitos, CA: IEEE Computer Society Press, November 1992, pp. 581–587.
36. M. Kishinevsky et al., *Concurrent Hardware: The Theory and Practice of Self-Timed Design,* New York: Wiley, 1994.
37. K.-J. Lin and C.-S. Lin, Automatic synthesis of asynchronous circuits, *Proc. ACM / IEEE Design Autom. Conf.,* Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 296–301.
38. P. Vanbekbergen et al., Optimized synthesis of asynchronous control circuits form graph-theoretic specifications, *Proc. Int. Conf. Comput.-Aided Design ICCAD,* Los Alamitos, CA: IEEE Computer Society Press, 1990, pp. 184–187.
39. M.-L. Yu and P. A. Subrahmanyan, A path oriented approach for reducing hazards in asynchronous design, *Proc. ACM / IEEE Design Automation Conf.,* 1992, pp. 239–244.

40. P. Vanbekbergen et al., A generalized state assignment theory for transformations on signal transition graphs, *Proc. Int. Conf. Comput.-Aided Design ICCAD,* Los Alamitos, CA: IEEE Computer Society Press, November 1992, pp. 112–117.

41. R. Puri and J. Gu, A modular partitioning approach for asynchronous circuit synthesis, *Proc. ACM / IEEE Design Autom. Conf.,* June 1994, pp. 63–69.

42. R. Puri and J. Gu, Area efficient synthesis of asynchronous interface circuits, *Proc. Int. Conf. Comput. Design ICCD,* Los Alamitos, CA: IEEE Computer Society Press, October 1994.

43. R. Puri and J. Gu, Asynchronous circuit synthesis with boolean satisfiability, *IEEE Trans. Comput.-Aided Design,* **14**: 961–973, 1995.

43a. A. Semenov et al., Synthesis of speed-independent circuits from STG-unfolding segment, in *Proc. ACM / IEEE Des. Automation Conf.,* pp. 16–21, 1997.

43b. J. Cortadella et al., Decomposition and technology mapping of speed-independent circuits using Boolean relations, in *Proc. Int. Conf. Comput.-Aided Des. ICCAD,* 1997.

43c. J. Cortadella et al., A region-based theory for state assignment in speed-independent circuits, *IEEE Trans. Comput.-Aided Des.,* **16**: 793–812, 1997.

43d. A. Kondratyev et al., Basic gate implementation of speed-independent circuits, in *Proc. ACM / IEEE Des. Automation Conf.,* pp. 56–62, 1994.

43e. E. Pastor and J. Cortadella, Polynomial algorithms for the synthesis of hazard-free circuits from signal transition graphs, in *Proc. Int. Conf. Comput.-Aided Des. ICCAD,* pp. 250–254, 1993.

44. J. Cortadella et al., Methodology and tools for state encoding in asynchronous circuit synthesis, *Proc. ACM / IEEE Design Autom. Conf.,* 1996.

45. C. Myers and T. H.-Y. Meng, Synthesis of timed asynchronous circuits, *Proc. Int. Conf. Comput. Design ICCD,* Los Alamitos, CA: IEEE Computer Society Press, October 1992, pp. 279–282.

45a. L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits,* Dordrecht: Kluwer, 1993.

45b. C. Ykman-Couvreur, B. Lin, and H. de Man, ASSASSIN: A synthesis system for asynchronous control circuits, Tech. rep., IMEC, Belgium, September 1994, user and tutorial manual.

46. C. A. R. Hoare, Communicating sequential processes, *Commun. ACM,* **21** (8): 666–677, 1978.

47. A. J. Martin, Compiling communicating processes into delay-insensitive VLSI circuits, *Distributed Comput.,* **1** (4): 226–234, 1986.

48. A. J. Martin, A synthesis method for self-timed VLSI circuits, *Proc. Int. Conf. Comput. Design ICCD,* Los Alamitos, CA: IEEE Computer Society Press, 1987, pp. 224–229.

49. S. M. Burns, *Automated Compilation of Concurrent Programs into Self-Timed Circuits,* Master's thesis, Pasadena, CA: California Institute of Technology, 1988.

50. S. M. Burns and A. J. Martin, Synthesis of self-timed circuits by program transformation, in G. J. Milne (ed.), *The Fusion of Hardware Design and Verification,* New York: Elsevier, 1988, pp. 99–116.

51. C. H. van Berkel, *Beware the isochronic fork,* Nat. Lab. Unclassified Report UR 003/91, Philips Research Lab., Eindhoven, The Netherlands, 1991.

52. A. J. Martin, The design of a self-timed circuit for distributed mutual exclusion, in Henry Fuchs (ed.), *Proc. 1985 Chapel Hill Conf. VLSI,* Computer Science Press, 1985, pp. 245–260.

53. A. J. Martin et al., The design of an asynchronous microprocessor, in Charles L. Seitz (ed.), *Advanced Research in VLSI: Proc. Decennial Caltech Conf. VLSI,* Cambridge, MA: MIT Press, 1989, pp. 351–373.

54. C. H. (K.) van Berkel et al., VLSI programming and silicon compilation, *Proc Int. Conf. Comput. Design ICCD,* Los Alamitos, CA: IEEE Computer Society Press, 1988, pp. 150–166.

55. E. Brunvand and R. F. Sproull, Translating concurrent programs into delay-insensitive circuits, in *Proc. Int. Conf. Comput.-Aided Design ICCAD,* Los Alamitos, CA: IEEE Computer Society Press, November 1989, pp. 262–265.

56. M. Rem, J. L. A. van de Snepscheut, and J. T. Udding, Trace theory and the definition of hierarchical components, in Randal Bryant (ed.), *Proc. 3rd Caltech Conf. VLSI,* Rockville, MA: Computer Science, 1983, pp. 225–239.

57. J. L. A. van de Snepscheut, *Trace Theory and VLSI Design,* vol. 200, Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1985.

58. J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits,* vol. 56 of CWI Tract, Centre for Mathematics and Computer Science, 1989.

59. J. C. Ebergen, *Arbitrers: An exercise in specifying and decomposing asynchronously communicating components.* Research Report CS-90-29, Computer Science Dept., Univ. of Waterloo, Canada, July 1990.

60. S. B. Furber et al., AMULET1: A micropipelined ARM, *Proc. IEEE Comput. Conf. COMPCON,* March 1994, pp. 476–485.

61. A. L. Davis, The architecture and system method of DDM-1: A recursively-structured data driven machine, *Proc. 5th Annu. Symp. Comput. Architecture,* 1978.

62. J. A. Tierno et al., A 100-MIPS GaAs asynchronous microprocessor, *IEEE Design & Test Comput.,* **11** (2): 43–49, 1994.

63. A. J. Martin et al., The design of an asynchronous MIPS R3000 microprocessor, *Proc. 17th Conf. Advanced Res. VLSI,* September 1997, pp. 164–181.

64. S. B. Furber et al., AMULET2e: An asynchronous embedded controller, *Proc. Int. Symp. Advanced Res. Asynchronous Circuits Syst.,* Los Alamitos, CA: IEEE Computer Society Press, April 1997.

65. R. Weiss, ARM researchers asynchronous CPU design, *Comput. Design,* 1995.

66. J. Yoshida, Sharp's processor beats the clock, *Electron. Eng. EE Times,* 1996.

67. P. Clarke, Startup pushes asynchronous chips towards mainstream, *Electron. Eng. EE Times,* October 1977.

68. T. Nanya et al., TITAC: Design of a quasi-delay-insensitive microprocessor, *IEEE Design & Test of Comput.,* **11** (2): 50–63, 1994.

69. R. F. Sproull, I. E. Sutherland, and C. E. Molnar, The counterflow pipeline processor architecture, *IEEE Design & Test of Comput.,* **11** (3): 48–59, 1994.

70. A. Davis, B. Coates, and K. Stevens, The Post Office experience: Designing a large asynchronous chip, *Proc. Hawaii Int. Conf. Syst. Sci.,* Los Alamitos, CA: IEEE Computer Society Press, January 1993, vol. 1I, pp. 409–418.

71. M. E. Dean, *STRiP: A Self-Timed RISC Processor Architecture,* PhD thesis, Stanford University, 1992.

72. K. van Berkel et al., A fully-asynchronous low-power error corrector for the DCC player, *Int. Solid State Circuits Conf.,* February 1994, pp. 88–89.

73. W. O. Budde et al., An asynchronous, high-speed packet switching component, *IEEE Design & Test Comput.,* **11** (2): 33–42, 1994.

RUCHIR PURI
IBM Thomas J. Watson Research Center

ASYNCHRONOUS MULTIPLEXING. See STATISTICAL MULTIPLEXING.