# Direct Implementation of Asynchronous Control Units

LEE A. HOLLAAR, SENIOR MEMBER, IEEE

*Abstract*—**The "one-hot" row assignment for asynchronous circuits, in which every row in a flow table has exactly one of the feedback variables that equals the value 1, provides a straightforward method for circuit synthesis. Once a flow table has been constructed, the state equations can be directly written, without requiring any procedure to ensure a race-free assignment. Furthermore, it can implement any arbitrary fundamental mode asynchronous circuit, not depending on a specific signaling protocol for its correct operation. An alternate view of one-hot asynchronous circuits is given, with a simple set-reset flip-flop for each state. Although this may seem excessive compared to implementations with encoded state variables, for many circuits their one-hot implementation is comparable in cost to other asynchronous implementations.**

**However, unless particular care is taken regarding the relative delays of the circuit components, the one-hot state assignment does not properly function when two states form a scale-of-two loop (where a state has another state as both its predecessor and successor). The reasons for this failure are discussed, along with implementations which solve the problem. The operation of the circuit will be detailed both under standard fundamental mode assumptions and when the inputs cause a transition before the circuit stabilizes. In most cases, the restrictions of the fundamental mode can be substantially relaxed. Finally, a number of expanded capabilities are presented, including a FORK/JOIN construct to allow two or more asynchronous activities to proceed simultaneously, and a subroutine capability, allowing the sharing of common sequences of states.**

*Index Terms*—**Asynchronous control, direct mapping algorithms, nonfundamental mode operation, "one-hot" state assignment, parallel operations, sequencer logical design, signaling protocols.**

## INTRODUCTION

ASYNCHRONOUS circuits offer substantial benefits in the design of digital control units or sequencers, particularly when many of the actions of the control unit are based on externally generated signals that are not guaranteed to be correlated to an available clock signal. Such signals are common in interfaces between central processors and their peripheral devices, particularly when connected to a bus designed for a wide variety of peripheral types and speeds. Often no special provisions for unsynchronized external inputs are made in the design of a clocked control circuit, which raises the possibility of incorrect circuit operation due to the circuit's flip-flops entering their metastable [1] state.

Special circuitry must be used to prevent metastable operation, or to rapidly force the flip-flops out of their metastable state. Since metastable behavior stems from violating the constraints of a digital circuit, such special circuitry generally requires analog techniques. One circuit for synchronizing an input to the control unit's clock, suggested by Fletcher [2], requires a *D*-type flip-flop and six additional gates, with two of the inverters being used as analog amplifiers.

Additionally, extra states must be added for the clocked control unit to idle while waiting for an external signal to occur. Each must be separated by only one state variable from the state entered when their external signal occurs to prevent incorrect transitions [2]. This further complicates the "simple" clocked implementation of a control unit with a number of external signals.

Alternatively, asynchronous design techniques can be used to implement the control unit. There is, obviously, no need for special circuitry to synchronize an external signal to an arbitrary clock. Nor are special idle loop states required, since the asynchronous control unit simply waits in a state until the desired external signal occurs, then immediately proceeds to the next state.

*Modular Implementation Methods:* Two methods have been employed for the implementation of asynchronous control units. The first is the use of specially designed modules which implement specific control activities (such as initiating an action or flow of control) [3]–[5]. However, this approach requires the use of specific signaling protocols between the control portion and the rest of the system (generally a variant of the four-cycle handshake) [6]. Although this system-wide view of the machine is important in producing a workable asynchronous machine implementation, the use of a specific protocol throughout may be difficult to achieve if the circuit being designed must interface with other devices with existing, but different, protocols, such as the channel or bus of a computer or a memory system.

For example, the *Q*-bus protocol used by the Digital Equipment Corporation on a number of their low-end processors [7] uses an asynchronous protocol which departs from the request/acknowledge of the four-cycle or two-cycle handshakes. Because it assumes upper bounds on the propagation delays on the bus and nominal times to perform certain simple actions, in some cases actions are taken without an acknowledgment of the previous action. While this provides a reliable, asynchronous method of transferring data, it does not directly match the protocols required by most modular techniques, and would be difficult for them to implement.

*Conventional Implementation Methods:* The second method is to design the control unit using the "conventional" methods detailed in a number of texts [8]–[10]. No specific protocol is required, although certain restrictions, such as the fundamental mode requirement that only one external input

Manuscript received September 17, 1981; revised January 12, 1982 and April 6, 1982.

The author is with the Department of Computer Science, University of Utah, Salt Lake City, UT 84112.

0018-9340/82/1200-1133$00.75 

can change at a time, and that the circuit must be stable when a change occurs [9], [11], can simplify the design process. Request/acknowledge protocols, such as the four-cycle handshake, can be easily implemented by regarding the request signals as the inputs to the control and the acknowledge signals as outputs, occurring when the circuit has performed a specific act or has stabilized.

Unfortunately, the application of conventional asynchronous design techniques to control units can become very difficult. Most techniques are oriented toward recognizers, which are characterized by a limited number of inputs and outputs (typically one or two inputs and a single output indicating that a particular pattern has been recognized) and a complex set of transitions between states. The most useful application of asynchronous circuits, however, are in sequencers with many inputs, most of which are not of interest at a given instant, and many outputs, with each state generally having a unique combination of the outputs. Many inputs can make flow tables unmanageable, since a column must be included for each *combination* of the inputs, and unique outputs for most states means that conventional state minimization techniques will be of little value.

A major task when using conventional asynchronous design techniques is to determine a state variable assignment with no critical races or undesirable output hazards. This is time-consuming for sequencers with many states, and does not guarantee the lowest cost implementation even when the assignment with the least number of state variables is found. Furthermore, any change in the specification of the control unit, particularly one which adds a state or changes a transition term, may necessitate a new state assignment [12] and the consequent total redesign of the circuit.

## The "One-Hot" State Assignment

An asynchronous circuit can be implemented directly from its state diagram, without the intermediate steps of flow table construction or determining a safe state variable assignment. This direct mapping implementation technique is based on the use of a "one-hot" code, where there is a state variable for each row in the flow table and only one state variable is a 1 whenever the circuit is stable. This technique was first discussed by Huffman [11] as the "one-relay-per-row" realization of an asynchronous sequential circuit. In addition to its ease of implementation, it had the benefit of only one relay being activated when the circuit was stable, reducing power consumption.

Since two state variables must change value during any state transition (the one associated with the current state changes from 1 to 0, while the next state's changes from 0 to 1), care must be taken to ensure that a critical race does not occur. This is done by making every transition a cycle by ensuring that the next state's variable is set first, then the current state's is reset (causing a transition sequence in the form 10–11–01). As an example, consider the circuit (described by Unger [8] on page 98), whose flow table and state diagram are shown in Fig. 1. It has two inputs, $x_1$ and $x_2$, no outputs, and five states.

The expressions for the state variables are given by Unger as

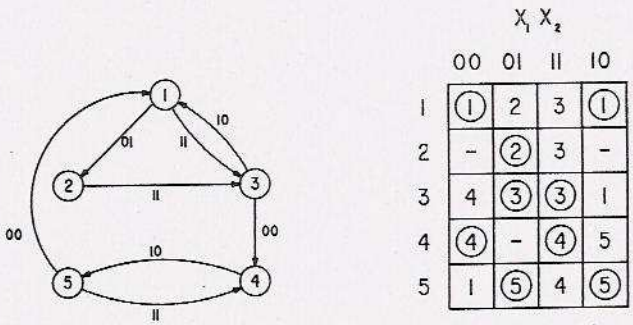



Fig. 1. Flow table and state diagram for Unger's example machine.

$$Y_1 = \bar{x}_1\bar{x}_2y_5 + x_1\bar{x}_2y_3 + y_1\bar{y}_2\bar{y}_3$$

$$Y_2 = \bar{x}_1x_2y_1 + y_2\bar{y}_3$$

$$Y_3 = x_1x_2y_1 + x_1x_2y_2 + y_3\bar{y}_1\bar{y}_4$$

$$Y_4 = \bar{x}_1\bar{x}_2y_3 + x_1x_2y_5 + y_4\bar{y}_5$$

$$Y_5 = x_1\bar{x}_2y_4 + y_5\bar{y}_1\bar{y}_4.$$

Each equation is of the form $Y_i = T_i + y_i\bar{H}_i$. $T_i$ is the *transition term*, consisting of the Boolean sum of all state transitions (current value of the state variables and inputs), or unstable total states, leading to state $i$. For example, the unstable total state in the lower left corner of the flow table in Fig. 1, which leads to state 1, has the value $\bar{x}_1\bar{x}_2y_5$. The remainder of each equation, $y_i\bar{H}_i$, is the *hold term*, which keeps state variable $i$ true until another state is entered. $H$ is the sum of the state variables for states whose predecessor is $i$; so, for the circuit of Fig. 1, the hold function for $Y_1$ is $y_1\bar{y}_2\bar{y}_3$, since state 1 can be followed by either state 2 or state 3.

### *An Alternate View*

The one-hot asynchronous machine implementation can alternatively be viewed as having a simple set-reset flip-flop for each state, which is set during a transition into the state, and which in turn resets its predecessor state flip-flop. To see that these are equivalent, consider the simple state diagram segment and its one-hot implementation given in Fig. 2. The segment consists of three states ($J$, $K$, and $L$) with transition inputs $R$, $S$, and $T$. Unlike the previous example, these inputs are shown in an unencoded form, rather than listing all combinations of the circuit inputs, since only particular input combinations are of interest at any one time. By examination of the state transitions, it is clear that, for example, the equation for state variable $K$ is $JS + K\bar{L}$, and this is implemented in Fig. 2 by gates 5–7, and 12.

The circuit of Fig. 2 can be easily transformed from the AND-OR form into its equivalent set-reset flip-flop form. Gates 6 and 7 actually form a flip-flop set by a 1 from gate 5 and reset by a 0 from gate 12. Because this particular flip-flop does not directly produce a complemented output, inverter 8 is necessary. This flip-flop and inverter can be replaced by the more conventional cross-connected NAND set-reset flip-flop, yielding the circuit of Fig. 3. The AND gate implementing the transition term is replaced by a NAND gate to provide the correct set polarity for the flip-flop.

In the following discussion, the cross-connected NAND

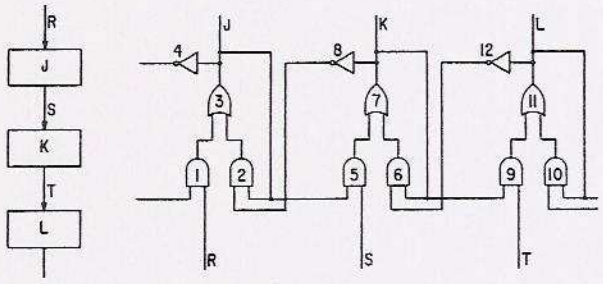



Fig. 2. One-hot circuit for a simple state diagram segment.

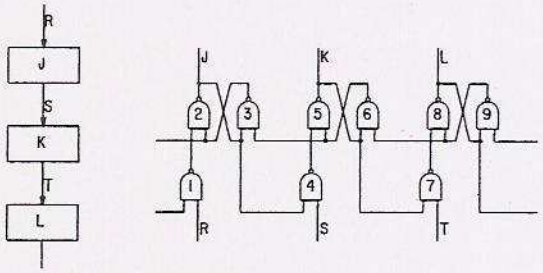



Fig. 3. A simplified version of Fig. 2's circuit.

symbol represents a set-reset flip-flop, and the transition term NAND represents the combinational logic necessary to set the flip-flop during a transition. These illustrate the topologies required by this direct mapping technique, although logic elements other than simple NAND gates could be used to implement them. In particular, it may be convenient to use a PLA-type array to generate the transition terms, particularly if the transition terms are more complex than a single input variable or to aid VLSI implementation by using a regular array. In this case, the transition NAND gate represents the portion of the PLA which generates the appropriate flip-flop set signal.

## Nonscale-of-Two Loop Circuit Operation

The transitions in a diagram can be divided into two classes. If a state has another state as both its immediate predecessor and successor, the transitions between the two states form a *scale-of-two loop;* all other transitions, regardless of whether they are part of any loop, are in the other class. In the state diagram of Fig. 1, transitions between states 1 and 3 form a scale-of-two loop, as do those between states 4 and 5. The transitions from state 1 to 2 or state 3 to 4 are not scale-of-two transitions. Scale-of-two loops are less frequent in sequencers than in recognizers, generally occurring when transferring data from one place to another using an interlocked technique. Because they are more common and operate in a more straightforward manner, nonscale-of-two loops will be discussed first.

### *Normal Circuit Operation*

Fig. 4 shows a slightly more complex state diagram segment than the one in Fig. 3, with state $A$ having two possible successors (state $B$ if the transition caused by $V$ is taken and state $C$ for the one caused by $X$). The fundamental mode restriction ensures that $V$ and $X$ cannot both become true when in state $A$ or during the transition to one of its successors. (The effects of violating this restriction will be discussed later.)

The same mapping as illustrated in Fig. 3 is used to produce the circuit in Fig. 4, with one flip-flop per state and a NAND gate representing the generation of the transition term. In Fig. 4, however, state $A$'s flip-flop enables two NAND gates, 4 and 7, since there are two possible transitions from state $A$. Flip-flop $A$ also has two reset signals, since it has two possible successors, each of which must be capable of resetting it to complete the transition. Because state $E$ has two predecessors, its flip-flop has two set inputs and produces the reset for two other flip-flops, $B$ and $D$.

The performance of the circuit can be readily determined. Looking at the transition from state $J$ to $K$ in Fig. 3, it is clear that it requires six gate delays from the time that transition term $S$ becomes true until no further changes in gate outputs occur and the circuit is stable. The path is from gate 4 to gate 5, then through gates 6, 3, 2, then back through gate 4. However, the result of the state transition, the true flip-flop output from the next state $K$, is valid after only two gate delays, and both flip-flops have reached their final values after only five gate delays. There is no requirement that the gate delays be uniform; in fact, the circuit will behave correctly as a fundamental mode circuit with arbitrary delays assumed for each gate.

### *Nonfundamental Mode Operation*

In the previous discussion, it was assumed that the asynchronous circuit inputs followed the requirements for fundamental mode operation: only one input can change at a time, and no change can occur unless the circuit is stable [8]–[10]. However, it may be impossible to ensure that these input requirements are met. While the one-hot implementation is not specifically designed for nonfundamental mode operation, it does permit substantially less stringent requirements than fundamental mode for correct circuit operation.

A variety of techniques have been proposed for designing asynchronous circuits whose inputs do not meet the requirements of fundamental mode. These generally require placing a number of constraints on the delays with the circuits [13], [14], or, if the delays are unbounded, using a special input protocol which places a spacer code between consecutive inputs [15], something which might not be possible for a given system. It appears that as long as gate delays remain reasonably uniform, the direct implementation technique does not need to use either of these approaches to correctly handle inputs which violate fundamental mode.

It is convenient to divide the inputs into three classes for any particular state. The first class consists of inputs which do not cause a transition from the particular state. Because the transition NAND gates to which these inputs are connected are not enabled, changes in their values can have no effect, and do not need to follow the requirements of fundamental mode.

*Sole Transitions from a State:* The second class of inputs are those which can cause a transition from a particular state, but that state has only one possible successor. As was mentioned above, the true flip-flop output for a state is valid two gate delays after the transition term into that state becomes

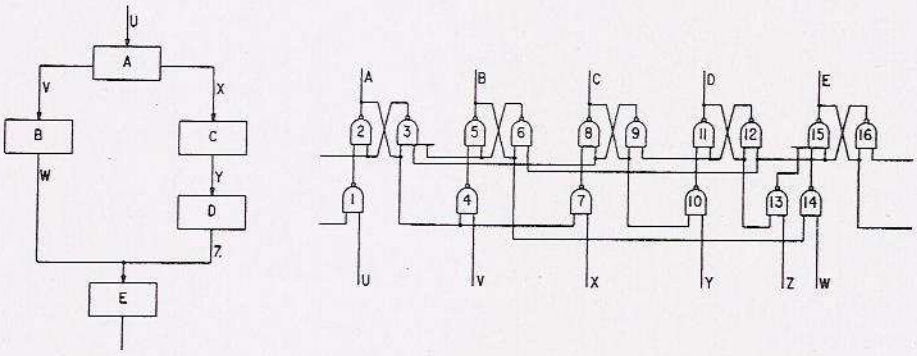



Fig. 4. State diagram segment with alternative successors and its implementation.

true. This true flip-flop output is also used to enable the transition NAND gate to the next state.

Assume that the circuit of Fig. 3 is in state $J$, transition term $S$ occurs, and $T$ is already true or becomes true shortly after $S$, but before the circuit can stabilize. Two gate delays after $S$ occurs, $K$ will become true and enable gate 7, which, since $T$ is already true, will generate a set signal for flip-flop $L$ after one more gate delay. Flip-flop $L$ will be set two gate delays after $K$ becomes true, and flip-flop $J$ will be completely reset three gate delays after $K$. Therefore, if the transition input from a state is already true when a state is entered, the next state will be entered after two gate delays and will overlap the time the original state is valid by three gate delays. This overlap is the same as for normal, fundamental mode operation.

Incorrect operation will only occur if a flip-flop is reset before it is able to successfully reset its predecessor, which will not occur if the gate delays are reasonably uniform. This failure can be prevented by disabling a transition until both its predecessor state is active and all predecessors to that state are inactive. For example, in Fig. 3, gate 7 would not only have state $K$ as an input, but also the complement of state $J$.

A problem can occur if a transition input becomes false after its transition starts but before the next state is entered. For example, if the circuit in Fig. 3 is in state $J$ and transition term $S$ occurs, the circuit will start the transition to state $K$. If $S$ remains true until state $K$'s flip-flop is completely set, no difficulties will result, since flip-flop $K$ will reset flip-flop $J$. The setting of flip-flop $K$ occurs three gate delays after $S$ becomes true and starts the transition; this is the minimal input pulse width required after a transition is enabled to ensure correct operation. The action of the circuit when the input pulse width is less than this depends on the switching characteristics of the gates, with the possibility of the state flip-flop entering a metastable [1] state due to a "runt" set pulse. This problem is eliminated, of course, if a demand/response protocol, such as the four-cycle handshake is employed, where an input cannot be removed until it is acknowledged.

*Alternative Successors to a State:* The final class of nonfundamental mode input is those which cause a transition from a particular state which has more than one successor. Improper operation can result if two inputs which can cause transitions occur almost simultaneously. For example, consider inputs $V$ and $X$ in Fig. 4. If both inputs occur at approximately the same time, flip-flops $B$ and $C$ will both be set, causing incorrect operation. Therefore, a more stringent requirement must be placed on inputs in this final class: when one occurs, the other must remain false until its transition NAND gate is disabled. This occurs after five gate delays, approximately the time for the circuit to become stable. This requirement can be reduced to one gate delay by a simple modification of the circuit: in Fig. 4, the output of gate 4 becomes an additional input to get 7 and the output of gate 7 an input to gate 4. This causes the first transition term to disable the transition NAND of the second.

If even this substantially reduced requirement cannot be met, a mutual exclusion circuit similar to the one described by Seitz [16] must be employed. This is a nondigital device which has an output for each input, and at most one output is active at any given time, indicating the input which became active first. The amount of time required to produce the output depends on how closely two inputs co-occur, with about a gate delay required if there is no co-occurrence and a much longer, undefined time if two inputs are simultaneous. The outputs of this mutual exclusion circuit then replace the previous inputs to the control unit.

*Summary of Nonfundamental Mode Operation:* In summary, as long as gate delays remain uniform the direct implementation technique operates well with inputs which violate the fundamental mode restrictions. If the input does not cause a transition from the current state, no restrictions are imposed. Inputs which can cause state transitions must have a pulse width of at least three gate delays and must be separated from other inputs which can cause a transition from a state by at least five gate delays. A minor modification to the circuit changes this separation to a single gate delay at the expense of an increase in the minimum pulse width. Alternatively, a mutual exclusion element can be included to remove the separation requirement altogether, at the expense of a delay in recognizing an input change.

## Scale-of-Two Loop Operation

Unfortunately, the implementation discussed above cannot be used for transitions involved in a scale-of-two loop. Consider the state diagram segment in Fig. 5, and its expected implementation. Transitions $B$ and $C$ form a scale-of-two loop. Assume that the circuit is in state $M$, and $B$ becomes true, causing the set input to flip-flop $N$ to become active. However, because the flip-flop for state $M$ is still set, the reset input for

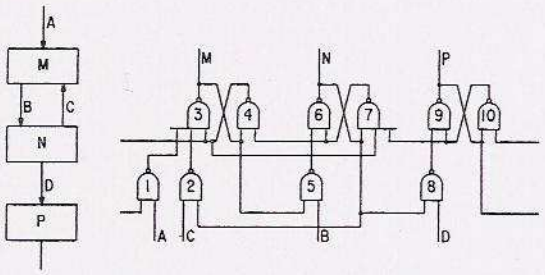



Fig. 5. A scale-of-two loop and its expected implementation.

flip-flop $N$ is also active, keeping it in an unstable state where both its true and complement outputs are 1. The circuit is hopelessly deadlocked waiting for state $M$ to be reset before state $N$ can reset it.

Although the particular failure mode is dependent on the implementation technique used, the problem with scale-of-two loops is inherent in the standard one-hot state assignment. The basic state transition sequence is 10–11–01 (showing only the two state variables involved in the transition), the first transition (from 10 to 11) caused by an input change, and the second (11 to 01) occurring because 11 is an unstable state which leads automatically to 01. For a scale-of-two loop only, both transitions 01–11–10 and 10–11–01 are possible. In this case, the unstable state 11 leads both to 01 and 10. It is clear that some mechanism is necessary to ensure that the transition from state 11 be to the correct successor.

An obvious solution to the problem is to convert all scale-of-two loops to scale-of-three loops, which, because three distinct unstable states are involved, do not produce incorrect behavior. Alternatively, special care can be taken to guide the transition from the unstable 11 state to the correct successor state, based on either the previous state or the input causing the transition. To see why this problem exists in the standard one-hot state assignment and why these solutions work, consider the one-hot equations for the state diagram segment in Fig. 5:

$$M = \cdots A + C N + M \overline{N}$$

$$N = B M + N \overline{M} \overline{P}$$

$$P = D N + P \cdots .$$

Looking at the equation for state $N$ given above ($B M + N \overline{M} \overline{P}$), there is a static 1-hazard [8]–[10], since there is no term in the expression which holds it true while $M$ switched from 1 to 0. To resolve this hazard without altering the equations, it is necessary that $\overline{M}$ becomes true before $M$ becomes false. For the implementation discussed above, the inverter used to produce $\overline{M}$ from $M$ must have negative delay, since, for a state to be successfully entered (its state variable latched by the flip-flop), it is necessary that the hold term of the state equation be true at the time the transition term becomes false. This is also true for $\overline{N}$ in the equation for $M$. This requirement for a negative-delay inverter can be solved instead by introducing additional delay in the $M$ input to the transition term, a standard technique for resolving a particular static hazard when the approximate delays for all other gates are known. This additional delay in the $M$ input provides a memory of state $M$ to guide the transition from the unstable state to the correct final state.

Alternatively, the static 1-hazard can be removed by introducing redundant implicants into the expressions for $M$ and $N$, ensuring that the state equation remains true throughout the transition regardless of the delays present. The redundant implicants necessary are simply the consensus [9] of the hold term and transition term involved in the scale-of-two loop. For state $N$ in the above equations, the consensus term is $B N \overline{P}$, so the equation for $N$ becomes $B M + N \overline{M} \overline{P} + B N \overline{P}$. The redundant implicant uses the input causing the transition to drive the circuit from the unstable state to the proper stable state. (In his discussion of the one-hot implementation, Unger does not discuss these inherent static hazards and the techniques necessary to eliminate them [8].)

Fig. 6 illustrates a properly functioning scale-of-two loop implementation using a redundant implicant to eliminate the static hazard. The expression for $N$ has been factored to give $B M + N \overline{P}(B + \overline{M})$, and a corresponding factoring was done for $M$. Comparing this equation to the original equation for $N$, the change necessary for proper scale-of-two operation is to replace the reset line from the other state involved in the loop with the OR of that reset line and the input which causes the transition to the state. In this case, the reset term is removed from the next state's flip-flop by the additional gate (for example, gate 12 for flip-flop $N$) whenever the input causing the transition that state is active.

*Scale-of-Two Nonfundamental Mode Operation*

The operation of the circuit in Fig. 6 for nonfundamental mode operation can be readily determined. The input condition of interest is when one of the scale-of-two loop transition inputs becomes true before the circuit has stabilized; it is the only case which differs from the previous discussion for nonscale-of-two segments. Like the nonscale-of-two case, an input can cause a transition from a state two gate delays after that state is entered, but five more gate delays are required for the circuit to stabilize. However, while another transition starting during this stabilization period did not cause incorrect behavior for the nonscale-of-two case, it can produce an anomaly for scale-of-two transitions.

Assume that, for the circuit in Fig. 6, the circuit is initially in state $M$ and both $B$ and $C$ become true simultaneously (or, since $C$ is not considered until state $N$ is entered, $C$ is already true when $B$ becomes true). This will cause gate 5 to set flip-flop $N$ while gate 12 removes the $N$'s reset signal. State $N$ becoming true along with transition term $C$ being already true will cause gate 2's output to become a 1 and gate 11 will inhibit the resetting of flip-flop $M$ by flip-flop $N$. As long as both transition terms $B$ and $C$ remain true, both flip-flops $M$ and $N$ will remain set. When either transition term becomes false, the circuit will enter the state which normally follows the transition term which remains true. For example, if $B$ were to go false after both $B$ and $C$ were true, state $M$ would be entered and the flip-flop for state $N$ would be reset.

This behavior is not unexpected, since, for example, state $N$ is entered two gate delays after $B$ becomes true, and state $M$ is re-entered two gate delays later. However, it requires six

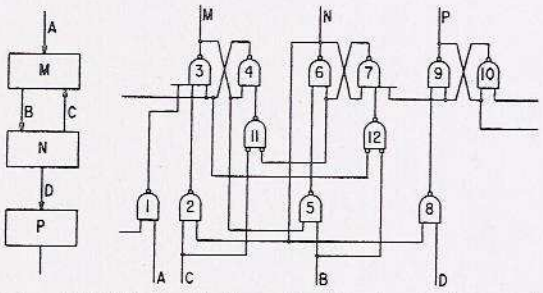



Fig. 6. A properly functioning scale-of-two loop implementation.

gate delays from the time $B$ becomes true for state $M$'s flip-flop to be completely reset, so state $M$ is re-entered two gate delays *before* it is completely exited. While this nonfundamental mode behavior is well defined, and proper operation continues when either of the two scale-of-two transition inputs are removed, and it can produce incorrect operation if some action must be performed in one of the states whenever the state is entered, since the flip-flop for the state remains set.

## Cost Considerations

It is simple to determine the cost of implementing a circuit using the direct mapping technique. States require either two or three gates each, depending on whether they are involved in a scale-of-two loop, and one gate is necessary for each transition. While the determination of the gate requirements for the minimal implementation of an asynchronous circuit is difficult, since it requires the complete design of the minimal implementation, some approximations can be made. One gate is required for each state variable ($\log_2$ of the number of states), and one gate is needed to decode each state. One gate is also needed for each transition, and one gate for each state to hold the circuit in a stable condition, although there is the possibility that the combinational circuit formed from these latter gates can be simplified. Often, extra state variables, and additional gates, may be necessary since no minimal assignment will produce race-free operation and hazards must be eliminated. Furthermore, the gates in a classical asynchronous design generally are not simple functions like AND or NAND, but have some inputs complemented and others uncomplemented, requiring additional inverters to synthesize the required gate.

For the simple five state recognizer of Fig. 1, without decoded states and with two scale-of-two loops, implementation requires 15 gates using conventional techniques, assuming that gates with arbitrarily complemented inputs are available. If a decoded output from each state is desired, as would be the case for most sequencers, this increase to 20 gates, and if only gates without complemented inputs (such as NAND's) are available, three additional inverters are necessary. The direct implementation requires 22 gates, of which 4 are necessary because of the two scale-of-two loops. Even in this case, the cost of the circuit implemented using the direct technique is comparable with a minimal conventional asynchronous design.

## Expanded Capabilities

A number of extensions to the basic direct mapping technique can be made to implement capabilities not normally found in classically implemented asynchronous circuits. These include a FORK/JOIN operation to allow concurrent activities and a subroutine capability to allow states to be shared between different portions of a machine.

### *Parallel Execution of Activities*

Although most modular techniques for implementing asynchronous circuits [3], [4] recognize the importance of allowing parallel execution of two or more sequences of states, the methods normally presented for developing fundamental mode asynchronous circuits are restricted to machines which can be in a single state at a time [2], [8]–[10]. At first glance, especially when viewed from the perspective of clocked machine design, this may not seem like a burdensome restriction, but it can substantially complicate the design of an asynchronous circuit and reduce its potential speed.

*Conventional Implementation Methods:* For example, consider a portion of the state diagram for a machine where two different activities, each having three states, can be executed in parallel without affecting the behavior of the machine, as illustrated in Fig. 7. Both activities are entered from state $A$ when input $P$ occurs and terminate in state $Z$. The first activity consists of states $B$, $C$, and $D$, while the second has states $E$, $F$, and $G$. Nothing is known regarding the relative times spent in each of these states. While it might seem attractive to produce a single string of states from two activities by combining the states pairwise (as would be done in a clocked implementation), this can result in substantially reduced performance.

For example, assume during one instance when the state diagram segment is entered that states $B$ and $G$ each take 1 time unit to complete, states $C$ and $F$, 10 time units, and states $D$ and $E$, 100 time units. Each activity then requires 111 time units to complete, which is also the time required to execute them both in parallel. However, if $B$ and $E$ are combined, it would take 100 time units for the new state to complete, the maximum time for any state in the combination. Similarly, it would take 10 units for the combination of $C$ and $F$, and 100 time units for $D$ and $G$, so a total of 210 time units would be required.

A solution is not to use a rigid pairwise combination of states, but to generate composite states so that a new state is entered whenever a state transition would have occurred in one of the parallel activities. For example, for the machine in Fig. 7, after input $P$ occurs, a transition will occur from state $A$ to a composite state with the combined outputs of states $B$ and $E$ (referred to in the following discussion as $B/E$). If input $R$ occurs before input $S$, the next composite state will be $C/E$, while if $S$ occurs first, it will be $B/F$. It is clear that if the relative durations of the states are not known when the machine is being designed, nine separate composite states will be necessary to ensure maximum performance. In general, if there are $i$ separate activities, and activity $j$ has $S_j$ states, $S_1 \times S_2 \cdots S_i$ composite states are required. For even a few simultaneous activities each with a small number of states, this can become a very large number, possibly more than can be conveniently handled using classical design techniques.

A more serious complication when using composite states stems from the fundamental mode requirement that input changes (and transitions) occur only when the circuit is stable. While it may be possible to control the inputs to a machine

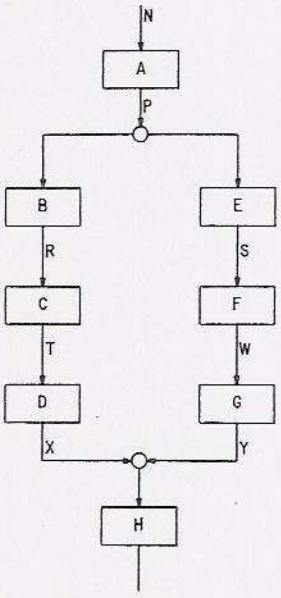



Fig. 7. A machine with two parallel activities.

without composite states to satisfy this requirement, it is virtually impossible to ensure that the requirement is met for composite states. By their very nature, the inputs which cause transitions in the different activities are independent. For example, in Fig. 7, inputs $R$ and $S$ have no correlation, since they are presumably generated by independent sources. It is entirely possible for input $S$ to occur while a transition caused by input $R$ is in progress, violating the fundamental mode restriction and causing incorrect circuit operation. Although additional logic, such as arbiters, might be added to help resolve this difficulty, this further complicates an already complex design.

*FORK and JOIN Operations:* It is convenient, when separate activities are to be performed concurrently, to think of the asynchronous control unit as dividing itself into a number of asynchronous submachines, one for each activity, and then recombining to form the original machine when the activities are all completed. This corresponds directly with the operators FORK and JOIN, suggested as scheduling primitives for multiprogramming applications [17]. As mentioned previously, this technique has been used in a number of modular implementation schemes.

It is possible to extend the direct implementation technique to handle FORK and JOIN operations with negligible changes to the design topology. Fig. 8 illustrates one possible implementation of the FORK operation using the direct technique. When the machine of Fig. 7 is in state $A$ and input $P$ occurs, a FORK transition (indicated by the small circle which splits the transition) to both states $B$ and $E$ should occur. This is implemented by having the transition gate for input $P$ set flip-flops for both states $B$ and $E$; state $A$ is reset when both $B$ and $E$ have been set, using gate 15. Gate 16 is used to provide an additional input to the transition gates for both $R$ and $S$, ensuring that transitions out of $B$ and $E$ will not occur until $A$ is completely reset. If the protocol specifications for the machine being implemented ensure that $B$ and $E$ will not be left before $A$ can be reset, gate 16 and the additional input to gates 7 and 12 can be removed. Furthermore, if the times necessary to correctly set flip-flops $B$ and $D$ are approximately the same,

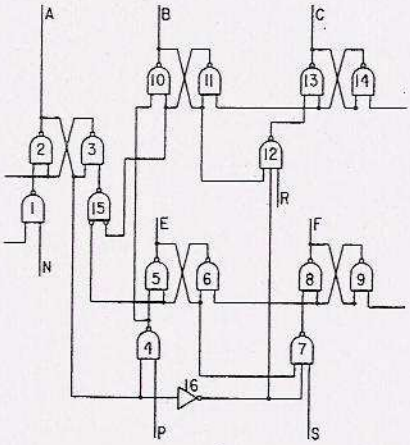



Fig. 8. Implementation of the FORK operation.

gate 15 can be removed and the complement output either $B$ or $E$ used to reset $A$.

After the FORK operation has occurred, the activities proceed as if each were a separate machine implemented using the direct techniques. Standard flow-of-control, allowing alternative states, can be used to implement a complex activity. It is even possible for an activity to perform another FORK, splitting the control even further.

The counterpart to the FORK operation is the JOIN, which combines the separate activities when they have all completed. Fig. 9 illustrates how the JOIN of Fig. 7 can be implemented. The flip-flop for state $H$ is set only when in state $D$ and input $X$ occurs and when in state $G$ and $Y$ occurs. Flip-flop $H$ is then used to reset both state $D$ and $G$'s flip-flops. Note that it is not a requirement that the activities joined by the operation be started by the same FORK operation.

This corresponds to schemes for implementing asynchronous, parallel activities as modeled using Petri nets [18]–[20]. Both use a token passing scheme to determine what states or places are currently active and what inputs or transitions are of interest, allowing the token to be split among a number of separate paths to provide parallel execution. The deciders of the Petri net correspond to the alternative successors of the direct implementation, where one of a group of possible successors or places are selected. The Petri net branch is similar to the fork operation in the direct implementation, and joins perform identical functions.

There are, however, substantial differences in the nature of the state diagram, augmented with FORK and JOIN operations, used by the direct implementation and a corresponding Petri net implementation. The Petri net implementations are based on using a particular signaling protocol, often the four-cycle handshake, with the firing of a labeled transition not only removing tokens from the input places and putting them in the output places, but sending a request signal to a specified execution unit and waiting for a response signal before actually placing the tokens. Decision between a number of events is implemented by having the execution unit return an alternative response if a particular condition occurs. The direct implementation method imposes no particular protocol, and, therefore, requires the explicit specification of actions (states) and events which cause actions to terminate and others to be initiated (transition terms).

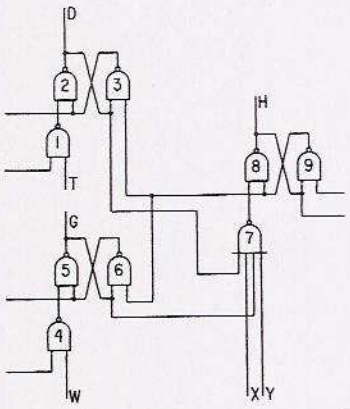



Fig. 9. Implementation of the JOIN operation.

A second difference from the token passing mechanism of Petri nets occurs when an input place is shared between two transitions. In this case, only one of the transitions will fire, since whichever one fires first will remove the token from the shared place, which, since it is now empty, will prevent the second transition from firing. This can be used to implement mutual exclusion, but requires the use of an arbiter to decide which transition will fire [16], [18]. No comparable facility exists in the direct implementation technique, although a particular protocol can be selected and an appropriate arbiter circuit included to provide mutual exclusion on a shared resource.

*Asynchronous Subroutines*

Just as subroutines have proved useful in simplifying the design of programs and reducing their storage requirements, common chains of states can be consolidated into "hardware subroutines" to simplify the design of asynchronous control units and reduce their hardware requirements. Asynchronous subroutines can be implemented using the direct technique with FORK and JOIN operations.

Fig. 10 illustrates how a subroutine consisting of states $X$, $Y$, and $Z$ can be activated from two different points in an extended state diagram. States $M$ and $R$ are normal sequential states, not part of parallel activities started by a FORK, so at most one can occur at the start and during the duration of the subroutine. If the machine is in state $M$ and input $B$ occurs, a fork operation to state $N$ and the start of the subroutine at state $X$ results. Processing proceeds in parallel in state $N$ and the subroutine $X/Y/Z$ until both state $N$ and input $C$, and state $Z$ and input $L$ occur, a JOIN is performed, and state $P$ is entered. Although only a single state $N$ is shown for the processing done in parallel with the subroutine, it can, of course, consist of as complex a sequence of states as is necessary.

It may be desirable to activate the subroutine but take no actions in parallel during its processing. This is illustrated by the processing starting at state $R$, entering the subroutine and state $S$ when input $F$ occurs. In this case, state $S$ is simply used as a flip-flop holding return address information, and generates no control signals. It has no inputs indicated on its transition, so the transition is performed immediately. When the subroutine $X/Y/Z$ completes, the JOIN operation occurs and state $T$ is entered; note that since $N$ was never activated during the sequencing from state $R$, $P$ cannot be activated at the JOIN operation.

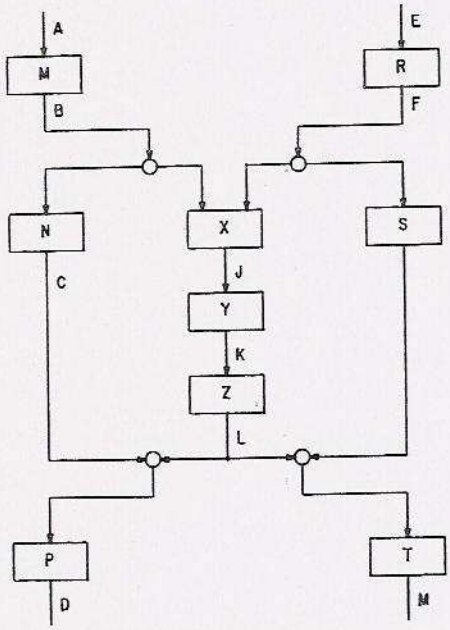



Fig. 10. An asynchronous subroutine for direct implementation.

*Synchronous Subcontrol*

Often there are activities of a control unit which are more suitable for clocked control rather than asynchronous implementations. For example, the stepping of a counter by a fixed number when only a step-by-one capability exists, requiring the sending of repeated control signals, is best done by a clocked control unless the counter provides a done signal or some other input can be used to moderate the asynchronous control's state transitions. A problem like this generally causes a control designer to choose a clocked control, even if an asynchronous implementation would offer other, and perhaps greater, advantages.

The solution is not to abandon the asynchronous control, but to force it when necessary into synchronous operation, returning to asynchronous operation when rigid synchronous operation is not required. This can be regarded as entering a portion of the state diagram where transitions occur based on a regular clock, rather than when an input occurs. The synchronous subcontrol can be implemented as an asynchronous circuit with the clock (and possibly its complement) as additional inputs, or by using any common method for clocked circuit implementation, although the method presented by Clare [21], using a $D$-type flip-flop per state, is probably most compatible with the direct asynchronous implementation technique, which also uses a flip-flop per state.

Exiting the synchronous portion of the machine presents no difficulties. The last state of the synchronous portion is simply used to set (or cause a transition to) the following asynchronous state. Entering the synchronous portion properly can be more difficult. What is required is to set the flip-flop corresponding to the first state of the synchronous portion from the asynchronous state which precedes it, and use that flip-flop's output to reset the asynchronous state's flip-flop. For proper operation,

all data setup time and minimum pulse widths must be observed for the clocked flip-flop selected. This requires the use of some form of synchronizer between the clock and the input which starts the synchronous portion, identical to that necessary when nonsynchronized inputs are used with a clock mode circuit. Depending on the elements used and the clock frequency, it may not be possible to implement such a synchronizer using strictly digital techniques. Fletcher [2] describes such a synchronizer, using inverters operating in their linear region as amplifiers, which he claims will remain metastable for only one clock period.

## Summary

The substantial advantages of asynchronous control unit design over clocked implementations, including a more natural, faster operation and ease of debugging, are well known. However, classical asynchronous design techniques are too complicated for many control designs. A method based on a direct mapping from a state diagram to a logic circuit, using only basic transformation rules, makes the design of an asynchronous implementation easier than the corresponding clocked design if there are a number of asynchronous inputs to the circuit. It can also be extended to handle FORK and JOIN operations and subroutines, to ease the design.

## Acknowledgment

The author wishes to thank D. Muller and J. Robertson, of the University of Illinois at Urbana-Champaign, for introducing him to asynchronous circuits in general, and the speed-independent techniques used in Illiac-2 in particular. Additional thanks go to G. Langdon and R. Haskin, both of IBM's San Jose Research Laboratory, for their many suggestions and help during the preparation of this paper, and to the referees, especially for their suggestions about solutions to the scale-of-two problem.

## References

[1] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," *IEEE Trans. Comput.*, vol. C-22, pp. 421–422, Apr. 1973.

[2] W. I. Fletcher, *An Engineering Approach to Digital Design.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

[3] S. M. Ornstein, M. J. Stucki, and W. A. Clark, "A functional description of macromodules," in *Proc. AFIPS Spring Joint Comput. Conf.*, 1967, pp. 337–355.

[4] B. J. Nordmann and B. H. McCormick, "Modular asynchronous design," *IEEE Trans. Comput.*, vol. C-26, pp. 196–207, Mar. 1977.

[5] *PDP16 Computer Designers Handbook*, Digital Equipment Corp., Maynard, MA, 1971.

[6] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980, pp. 218–262.

[7] *PDP11 Bus Handbook*, Digital Equipment Corp., Maynard, MA, 1979.

[8] S. H. Unger, *Asynchronous Sequential Switching Circuits.* New York: Wiley-Interscience, 1969.

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

[10] F. J. Hill and G. R. Peterson, *Introduction to Switching Theory and Logical Design*, 3rd ed. New York: Wiley, 1981.

[11] D. A. Huffman, "The synthesis of sequential switching circuits," in *Sequential Machines, Selected Papers*, E. F. Moore, Ed. Reading MA: Addison-Wesley, 1964, pp. 3–62. Reprinted from *J. Franklin Institute*, vol. 257, no. 3, pp. 161–190, Mar. 1954, and no. 4, pp. 275–303, Apr. 1954.

[12] G. G. Langdon, *Logic Design—A Review of Theory and Practice.* New York: Academic ACM Monograph, 1974.

[13] A. D. Friedman and P. R. Menon, "Synthesis of asynchronous sequential circuits with multiple-input changes," *IEEE Trans. Comput.*, vol. C-17, pp. 559–566, June 1968.

[14] S. H. Unger, "Self-synchronizing circuits and nonfundamental mode operation," *IEEE Trans. Comput.*, vol. C-26, pp. 278–281, Mar. 1977.

[15] D. B. Armstrong, A. D. Friedman, and P. R. Menon, "Design of asynchronous circuits assuming unbounded gate delays," *IEEE Trans. Comput.*, vol. C-18, pp. 1110–1120, Dec. 1969.

[16] C. L. Seitz, "Ideas about arbiters," *Lambda*, pp. 10–14, Quarter 1980.

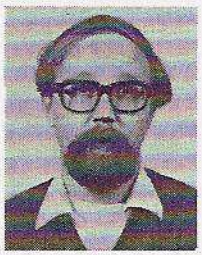
[17] M. E. Conway, "A multiprocessor system design," in *Proc. 1st AFIPS Fall Joint Conf.*, Nov. 1963, pp. 139–146.

[18] S. S. Patil and J. B. Dennis, "The description and realization of digital systems," in *Proc. IEEE COMPCON '72*, 1972, pp. 223–226.

[19] T. Agerwala, "Putting Petri nets to work," *Computer*, pp. 85–93, Dec. 1979.

[20] D. Misunas, "Petri nets and speed independent design," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 474–481, Aug. 1973.

[21] C. R. Clare, *Designing Logic Systems Using State Machines.* New York: McGraw-Hill, 1973.



**Lee Hollaar** (S'66–M'69–SM'79) received the B.S. degree in electrical engineering from the Illinois Institute of Technology, Chicago, in 1969, and the Ph.D. degree in computer science from the University of Illinois, Urbana-Champaign, in 1975.

He has been an Associate Professor of Computer Science at the University of Utah, Salt Lake City, since 1980. Prior to that he was a Graduate Research Assistant, and then an Assistant Professor of Computer Science and Senior Research Engineer for the Computing Services Office at the University of Illinois at Urbana-Champaign. He has been active in the development of hardware and software systems to support very large full-text information retrieval systems for over ten years, and is the principal inventor of a specialized processor system for merging lists of entries from an index file. In addition, he is the co-inventor of a new class of finite state automaton for the rapid searching of text stored on a rotating memory system. His research interests include high level VLSI design automation systems and broad-band communications networks. In the past he was Program Chairman for the Fourth Workshop on Computer Architecture for Non-Numeric Processing and serves on that Workshop's Steering Committee.

Dr. Hollaar is a member of the Association for Computing Machinery and the American Institute of Aeronautics and Astronautics. He is also a Registered Professional Engineer (control systems) in California.