Asynchronous state machines challenge digital designers

Part 1 of this 2-part series provided a brief refresher of the basic theory of state machines and gave two detailed examples of synchronous state-machine design with common PLDs. This article, part 2, continues with an example of a more difficult, asynchronous state machine. Part 2 also gives some background information on state-machine software packages and details a PLD whose architecture suits large state machines.

Stan Kopec, *Altera Corp*

Implementing state-machine designs in programmable-logic devices (PLDs) can solve some irksome control-logic design problems. If you elect to perform asynchronous state-machine design, however, be advised that it's more difficult than its synchronous counterpart. You should embark on an asynchronous state-machine design only when your application gives you no other choice.

Asynchronous state machines, by definition, will respond to any allowable input. Thus, they're susceptible to interference, noise, and glitches. Further, they require you to pay close attention to state assignment. Although careful state assignment is helpful but not critical in synchronous state-machine design, in asynchronous state machines it's crucial.

What's more, asynchronous designs are very sensitive to mismatched delays, races, and hazards. A race is a series of successive states that occurs during a state transition. A hazard occurs whenever more than one state variable changes at a time, resulting in unwanted transient states or potential decoding-logic glitches.

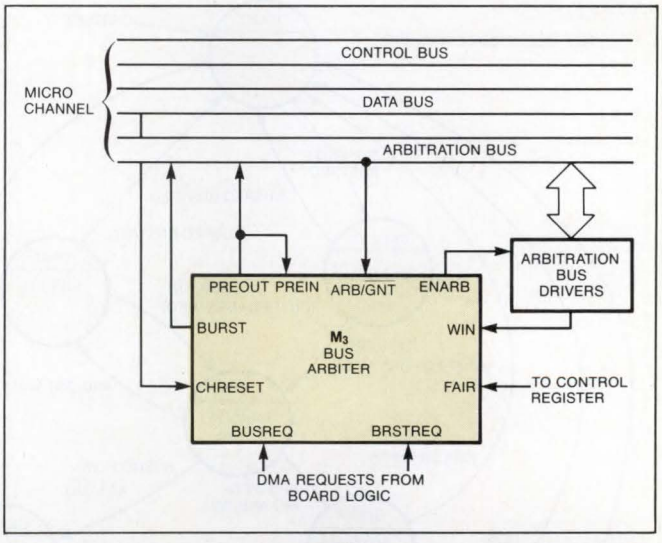

***Fig 1—This bus arbiter** for the IBM PS/2 Micro Channel bus accepts DMA requests from external devices and then strives to gain control of the bus.*

The problem of asynchronous state-machine design is evident in **Fig 1,** which shows a bus arbiter for the IBM Micro Channel (the bus for the company's PS/2 computers). The arbiter, M_3, has eight states. **Fig 2a** shows the state diagram for the bus arbiter. The inputs are ARB/$\overline{\text{GNT}}$, PREIN, WIN, CHRESET, FAIR,

If you're implementing an asynchronous state machine in which the input-to-output delays are critical, you may want to put the intermediate states to work.

BRSTREQ, and BUSREQ; the outputs are PREOUT, BUSGNT, BURST and ENARB.

The arbiter operates as follows. Peripheral logic asserts the BUSREQ (single-cycle) or BRSTREQ (block-transfer) lines to request use of the Micro Channel. The arbiter state machine (M_3) responds by asserting PREOUT to the Micro Channel. In response, the PS/2's μP asserts ARB/$\overline{GNT}$, indicating that arbitration has begun. M_3 asserts ENARB to place its arbitration priority on the bus. If the bus value matches its priority when ARB goes low, the arbiter has won the bus, and it proceeds to transfer data after asserting BUSGNT. If the priorities do not match, the arbiter has lost and must wait for another arbitration cycle. During block transfers, the arbiter asserts BURST.

Because the state variables in asynchronous machines are sensitive to glitches, changing only one state variable at a time proves to be a good rule of thumb. When you look at M_3, you might assume that it requires only three state variables because it has eight states. In fact, M_3 requires six state variables to satisfy all the state adjacencies. You determine adjacencies by listing each state with the states connected to it by arrows in the state diagram. To make adjacent states differ by only one state variable, you may need to insert intermediate states **(Fig 2b).**

In an asynchronous-machine design, you should decode invalid state-variable combinations to make the design more reliable. If the logic decodes an invalid state, it should reset the machine to an IDLE condition or an ERROR state. Thus, even if the machine misbehaves, at least it can't run amok.

Fig 3 (pg 184) shows the transition equations for M_3. M_3 is a Moore design. Its p-term requirements range from two to 14. If you used PLDs having eight p-term

Text continued on pg 184

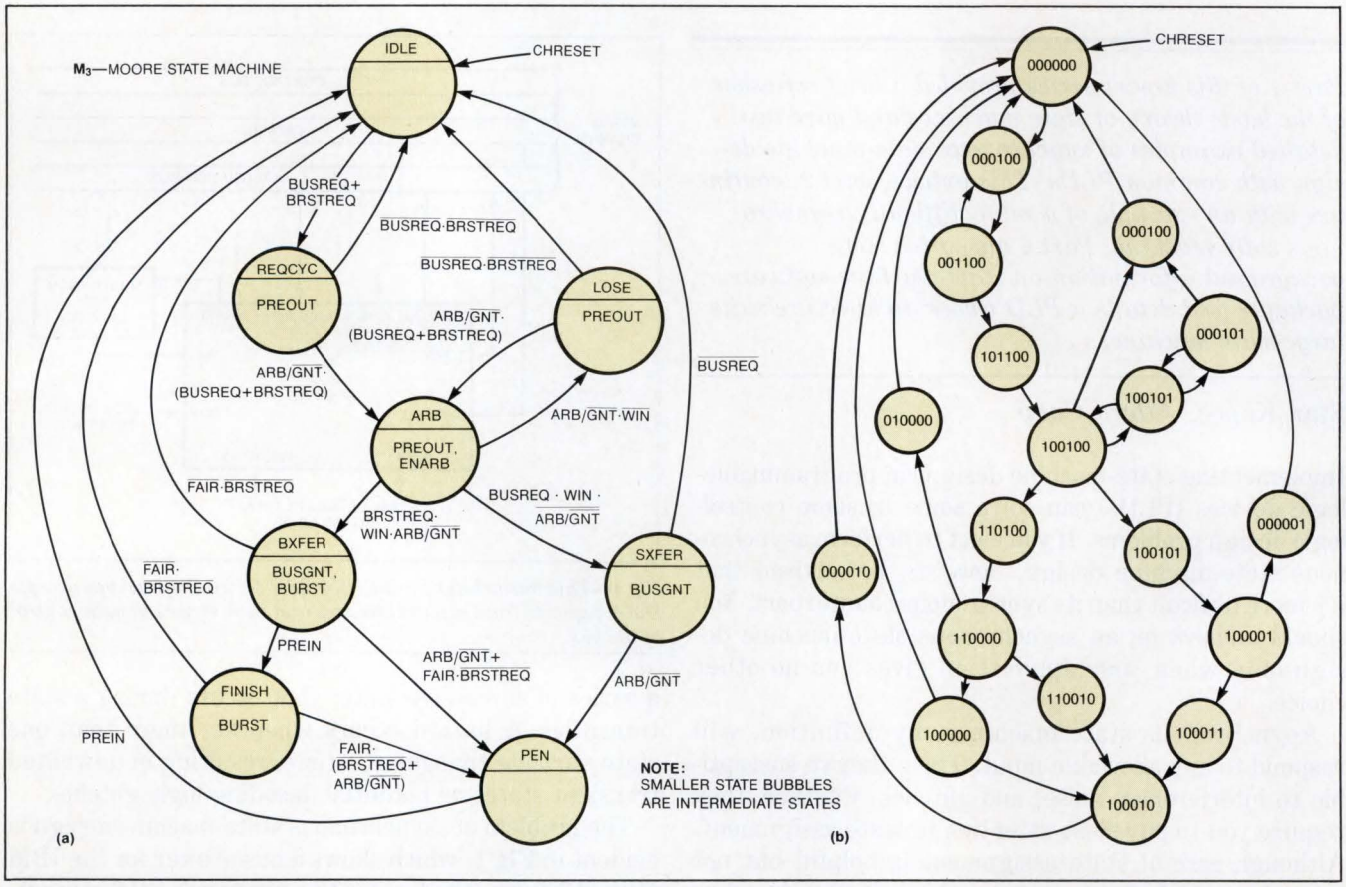

Fig 2—The state-transition-diagram *in* ***a*** *is the formal definition of the bus-arbitration handshake for the IBM Micro Channel bus. The diagram in* ***b*** *has an extra intermediate state so that the M_3 state machine will operate without glitches. Note that M_3 requires six state variables, instead of only three, to specify the eight required states. The extra variables are needed to satisfy the requirement that each state transition change only one state variable.*

State-machine software speeds design

State machine and general-purpose PLD-design software can automate many of the rote tasks associated with state-machine design. **Fig A** shows the major blocks of such a package. Because some of these software packages minimize logic functions—for example, by selecting deMorgan's inversion where appropriate—they can save you considerable time and effort. Some state-machine packages will also try different flip-flop types in a quest for the minimal implementation (your target PLD must have these flip-flop types, of course).

Some of these software packages also allow high-level functional descriptions. For example, high-level syntax for state machines means that you can enter statements for transition specifications rather than transition equations. By automatically generating the transition equations from the high-level description, the software packages save you much error-prone rote work as you try out different state assignments.

The software tools can further aid in your design effort by automatically reducing these initial equations to a minimal form. JEDEC-file assembly from these minimal equations is a straightforward translation.

These packages also provide a simple functional simulation—or, less frequently, a timing simulation—from the JEDEC-file description of your state machine. You can typically compile a state-machine design and simulate it on your PC in minutes with any of this software. This relatively rapid iteration cycle is of particular value when you formulate your machine design, because complex control flows are prone to designer error. Seeing your machine's operation in an interactive simulation environment allows you to work the bugs out quickly before you go to the lab to make a breadboard.

Existing PLD-design tools don't handle tasks such as state-variable assignment for you. Neither do they detect or report races and hazards (asynchronous state-machine designers, watch out!). Mealy-to-Moore conversion is also left to you. These areas are where the art of state-machine design begins.

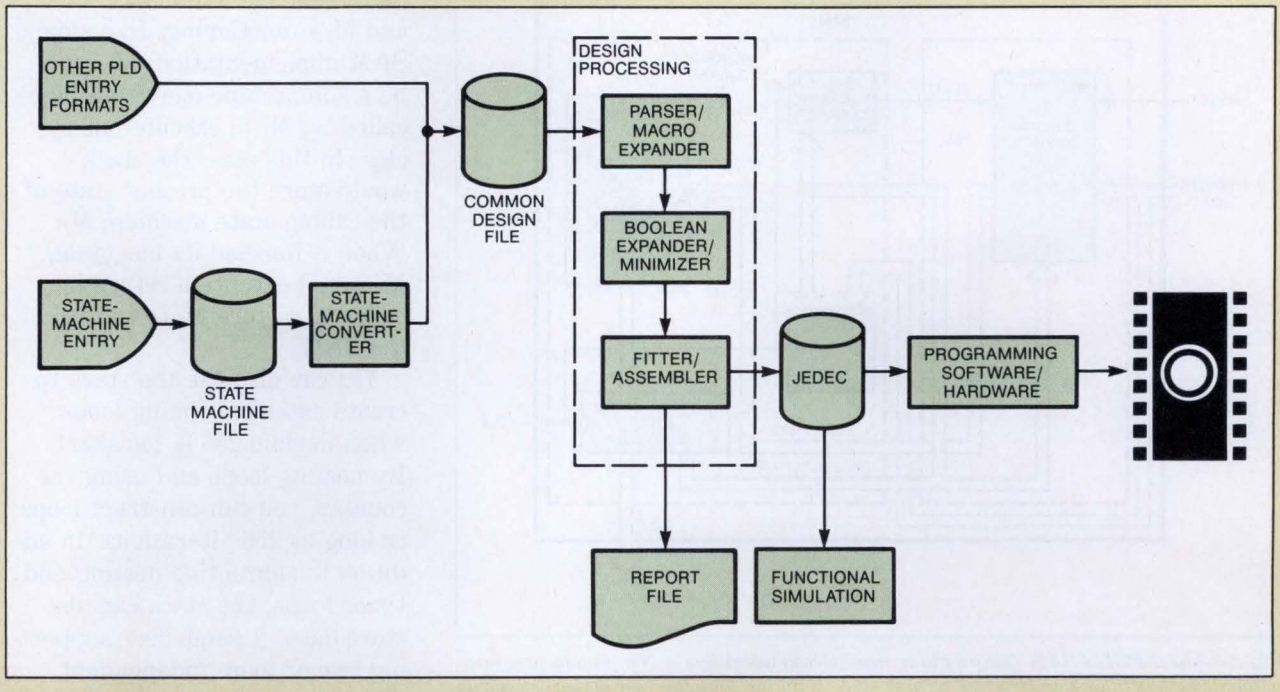

Fig A—State-machine and general-purpose PLD-design software *can automate many of the rote tasks associated with state-machine design.*

New PLD architecture suits large state machines

The EPS448 SAM (stand-alone microsequencer) is a 28-pin, CMOS EPROM-based chip (**Fig A**). It has a 768-p-term programmable-logic block for selecting one of four state-machine branches per state. It also includes 16k bits of microcode/state EPROM, an 8-bit loop counter, and a 15-level stack.

Functionally, you can consider the device to be a microcoded, instruction-based engine under the direction of a master programmable-logic block that determines branches.

The SAM's architecture implements a synchronous Moore state machine. A SAM can implement state machines with as many as 448 states. The device's 36-bit×448-word microprogram/state memory is organized into two sections: One section is one word wide for straight-line, unconditional sequences, and one section is four words wide for branching sequences.

Each microcode word has fields for the state of the device's outputs, a 3-state enable bit, an op code, and arguments for the op code. The arguments for the op code are the location in the microprogram memory of the next state to be jumped to, and, optionally, constants.

The branch-logic block combines the device's eight inputs and its present state (one of two 8-bit fields in the currently addressed word in the microprogram/state memory) to select one of four next states from among those in a branching-sequence memory location. Further, the SAM is unlike conventional PLDs in that its hold-state transitions consume no p-terms.

True to its microcoded architecture, the SAM can also perform functions more akin to those of a μP than to those of a classical state machine. For example, it can nest subroutines and execute timing loops. With the 8-bit, 256-state loop counter, in combination with the stack, you can generate arbitrarily long output-signal durations.

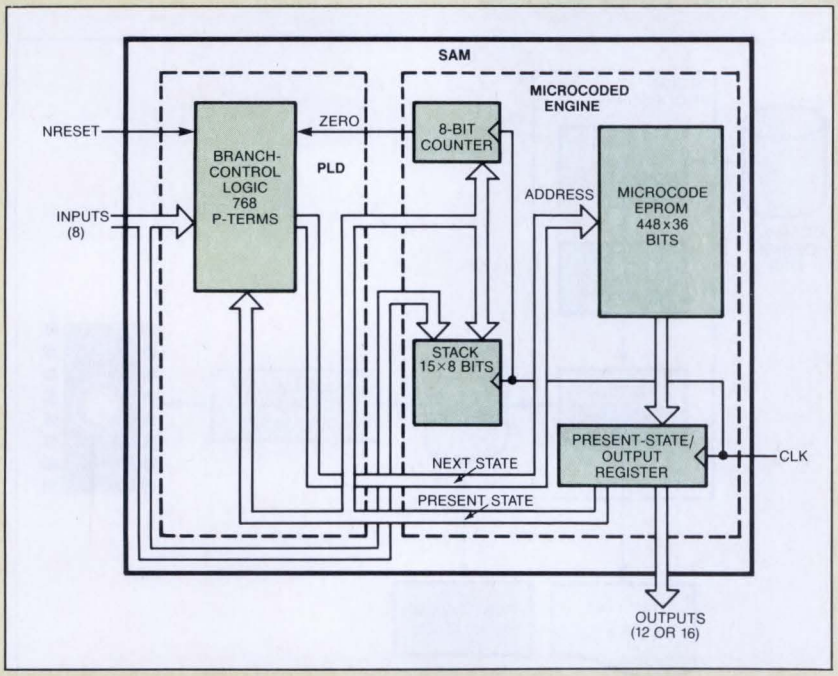

Fig A—The EPS448 SAM (stand-alone microsequencer) *has a 768 p-term programmable-logic block for selecting one of four state-machine branches per state. Operating as a bit-slice processor does, the SAM also includes 16k bits of microcode/state EPROM, an 8-bit loop counter, and a 15-level stack.*

The SAM has a primitive instruction set with which you can invoke the counter and stack at required points in the state-machine flow. The instructions include call, return, push counter, load counter, decrement, and test counter. You can use these instruction-based operations in conjunction with the programmable-branch logic.

The counter in the SAM has several potential uses in applications such as the synchronous DMA controller (M_1) and the synchronous bus controller (M_2) detailed in part 1 of this series (**Ref 1**). A bus-timeout function could force the termination of a bus cycle when READY does not occur within a specified time. If block data transfers involve a fixed-length (nonprogrammable) block of data, the counter could replace the external byte counter. To limit bus hogging, M_1 might transfer 16-byte subblocks, release the bus, and then request it again after some interval.

The stack on SAM could be used similarly to enhance M_1's and M_2's functioning. In a single-SAM implementation, M_2 could be a submachine (slave machine) called by M_1 to execute bus cycles. In this case, the stack would store the present state of the calling state machine, M_1. When it finished its bus cycle, M_2 would execute a return in order to restore M_1 to control of the SAM.

You can also use the stack to create extended timing loops when modulo 256 is too short. By nesting loops and using the counter, you can construct loops as long as 256^{15} iterations. In addition to subroutine nesting and timer loops, the stack can also store data. A sequencer supporting two or more independent DMA channels could use the stack for the temporary storage

and swapping of channel parameters, for example.

You can use multiple SAMs in parallel to increase the output-line count or to implement multichip controllers. Like memory chips, the architecture lends itself to modular expansion in both output width and microprogram depth.

Given the complexity of its architecture and the size of the design problems it addresses, the SAM requires high-level design software. The vendor's SAM+ software package can automatically compile your machine description from a text file. You can use high-level IF statements for state-machine transitions in conjunction with assembly-level SAM instructions that use counter and stack. You specify state outputs as vectors without worrying about logic or p-terms. **Fig B** shows a SAM specification for M_1.

Reference

1. Kopec, Stan, "State machines solve control-sequence problems," *EDN*, May 26, 1988, pg 177.

```
                SAM M1 DESIGN
PART:           EPS448
INPUTS:         BC0,STARTX,GRANT,DREQ,IO,SRC8,DST8,DONECYC
OUTPUTS:        REQBUS, XDONE,DECB1,DECB2,DECS1,DECS2,DECD1,DECD2, R16S,
                R16D,R8S,R8D,SWAP

PROGRAM:

A: [00 00 00 00 0000 0] JUMP B;

B: IF STARTX*IO' THEN [10 00 00 00 0000 0] JUMP D;
     ELSEIF STARTX*IO THEN [00 00 00 00 0000 0] JUMP C;
     ELSE [00 00 00 00 0000 0] JUMP B;

C: IF DREQ THEN [10 00 00 00 0000 0]JUMP D;
     ELSE [00 00 00 00 0000 0]JUMP C;

D: IF GRANT*SRC8' THEN [00 01 01 00 1000 0]JUMP I;
     ELSEIF GRANT*SRC8 THEN [00 01 10 00 0010 0]JUMP E;
     ELSE [10 00 00 00 0000 0]JUMP D;

E: IF DONECYC*DREQ THEN [00 00 10 00 0010 1]JUMP F;
     ELSE [00 01 10 00 0010 0]JUMP E;

F: IF DONECYC THEN [00 00 00 01 0100 0]JUMP G;
     ELSE [00 00 10 00 0010 1]JUMP F;

G: IF DONECYC*BC0 THEN [01 00 00 00 0000 0]JUMP A;
     ELSEIF DONECYC*BC0' THEN [00 00 00 00 0000 0]JUMP H;
     ELSE [00 00 00 01 0100 0]JUMP G;

H: IF DREQ THEN [00 00 10 00 0010 1]JUMP E;
     ELSE [00 00 00 00 0000 0]JUMP H;

I: IF DONECYC*DST8 THEN [00 00 00 10 0001 0]JUMP J;
     ELSEIF DONECYC*DST8' THEN [00 00 00 01 0100 0]JUMP M;
     ELSE [00 01 01 00 1000 0]JUMP I;

J: IF DONECYC*DREQ THEN [00 00 00 10 0001 1]JUMP K;
     ELSE [00 00 00 10 0001 0]JUMP J;

K: IF DONECYC*BC0 THEN [01 00 00 00 0000 0]JUMP A;
     ELSEIF DONECYC*IO*BC0' THEN [00 00 00 00 0000 0]JUMP L;
     ELSE [00 00 00 10 0001 1]JUMP K;

L: IF DREQ THEN [00 01 01 00 1000 0]JUMP I;
     ELSE [00 00 00 00 0000 0]JUMP L;

M: IF DONECYC*IO*BC0' THEN [00 00 00 00 0000 0]JUMP L;
     ELSEIF DONECYC*IO'*BC0' THEN [00 01 01 00 1000 0]JUMP I;
     ELSEIF DONECYC*BC0 THEN [01 00 00 00 0000 0]JUMP A;
     ELSE [00 00 00 01 0100 0]JUMP M;
```

Fig B—This high-level SAM specification *for the M_1 bus arbiter from part 1 of this series shows the proprietary IF-statement entry format for state transitions. To program for the chip's counter and stack, you must use assembly language.*

Because some PLD-design software packages perform minimization of logic functions, they can save you considerable time and effort.

macrocells, you would have to cascade macrocells to handle some of the state variables. A variable p-term device (22V10, EP1210) or a PLD with p-term redistribution (EP512) could handle this requirement directly. PLA devices would also provide a good result, because M_3 requires a total of 40 or fewer p-terms.

You implement asynchronous-state-machine state variables in combinatorial macrocells. Inserting intermediate states has the effect of adding another logic-array delay to your state transitions. **Fig 4** shows the PLD timing model for an asynchronous state machine. Note that the change in Q_0 takes the machine to an intermediate state and that this transition then triggers a change in Q_1 before the state machine makes the transition to the final state. This cascading means that a double state transition is actually occurring, and the extra T_{ARRAY} delay shown in the FREQUENCY equation is the result.

Delays for outputs from asynchronous machines are referenced to the appropriate inputs. Outputs typically become valid within one propagation delay (T_{PD}) of a new state's stabilizing. Because no central clock exists, you can't register outputs to minimize skews, as you can in synchronous machines.

If you're implementing an asynchronous state machine in which the input-to-output delays are critical, you may want to put those intermediate states to work. Normally, in a Moore machine, outputs are associated

```
            M3 State Assignment and Transition Equations

State      State Variables
          Q5 Q4 Q3 Q2 Q1 Q0
-------------------------
IDLE         000000
REQCYC       001100
ARB          100100
LOSE         000101
BXFER        110000
SXFER        100001
FINISH       100000
PEN          100010
------------------
INTER-       000100
   MEDIATE   101100
      |      100101
      |      000001
      V      100011
             110010
             000010
             010000
             110100

Q0 = (10010X*ARB/-GNT'*WIN' + 000101* (BUSREQ + BRSTREQ) +
     10010X*BUSREQ*WIN*ARB/-GNT + 100001)*CHRESET'          (5 TERMS)

Q1 = (1100X0*ARB/-GNT*FAIR*BRSTREQ + 100010 + 1000X1*ARB/-GNT)*CHRESET'
                                                           (3 TERMS)

Q2 = (000X00*(BUSREQ+BRSTREQ) + X01100 + 100100 +
     100101*ARB/-GNT'*WIN' + 100101*ARB/-GNT*(BUSREQ+BRSTREQ) +
     000101)*CHRESET'                                      (8 TERMS)

Q3 = (00X100*(BUSREQ+BRSTREQ))*CHRESET'                    (2 TERMS)

Q4 = (1X0100*BRSTREQ*WIN*ARB/-GNT + 110000*FAIR'*BRSTREQ' +
     110000*(ARB/-GNT+FAIR*BRSTREQ') + 110000*PREIN')*CHRESET'
                                                           (5 TERMS)

Q5 = (X01100*(BUSREQ+BRSTREQ) + 100100*ARB/-GNT'*WIN' +
     10010X*ARB/-GNT*(BUSREQ+BRSTREQ) + 1X0100*BRSTREQ*WIN*ARB/-GNT +
     1100X0*(ARB/-GNT+FAIR*BRSTREQ') + 1X0000*(FAIR+BRSTREQ) +
     100010*PREIN + 100101*BUSREQ*WIN*ARB/-GNT' +
     1000X1*BUSREQ + 110000*PREIN)*CHRESET'                (14 TERMS)
```

Fig 3—Given the state assignments in Fig 2b, *these transition equations result for M_3. M_3 is a Moore machine.*

Existing PLD-design tools don't handle such tasks as state-variable assignment for you.

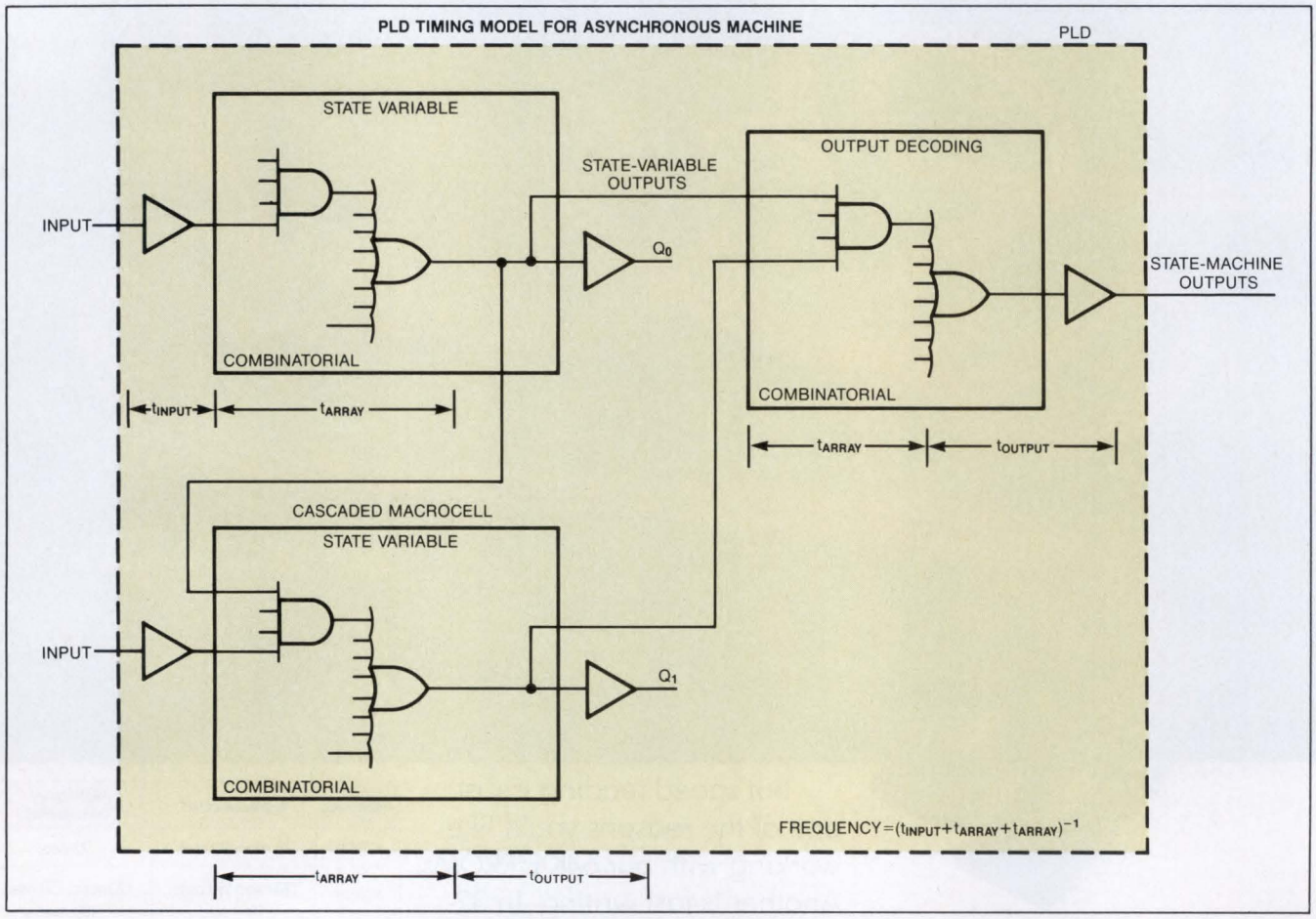

Fig 4—When you use this PLD timing model to calculate *assignments in Fig 2b, these transition equations result for M_3. M_3 is a Moore machine.*

with primary states. If you find that an output can't tolerate the added delay of the intermediate state, you can activate the output at both the intermediate state and the primary state. This assignment eliminates the added logic-array delay in generating the output, so it effectively performs an output look-ahead. Be careful, though: If you use an intermediate-state code in more than one place (as in **Fig 2b**'s state diagram of M_3, which has state 100101 in more than one place), you'll need to decode the intermediate state *and* the inputs to guarantee that the output will be correct.

Given this model, you can see that a PLD that implements M_3 will be able to operate at

$$\text{Frequency}=1/T_{INPUT}+(2\times T_{ARRAY}).$$

Therefore, a typical PLD in which $T_{INPUT}=5$ nsec and $T_{ARRAY}=20$ nsec will operate at approximately 22 MHz. PLDs having a faster speed grade would give correspondingly faster frequencies. **EDN**

Author's biography

Stanley Kopec is manager of product planning for programmable logic at Altera Corp in Santa Clara, CA. He has been with Altera for three years. Prior to joining Altera, Stan worked for Exel Microelectronics, where he was in charge of μP-peripheral development. He holds a BSEE from the State University of New York at Buffalo and an MSEE from the University of Illinois. In his spare time he enjoys racquetball, skiing, and reading.

Article Interest Quotient (Circle One)
High 482 Medium 483 Low 484