



INTRODUCTION

State machine designs are widely used for sequential control logic, which forms the core of many digital systems. State machines are required in a variety of applications covering a broad range of performance and complexity; low-level controls of microprocessor-to-VLSI-peripheral interfaces, bus arbitration and timing generation in conventional microprocessors, custom bit-slice microprocessors, data encryption and decryption, and transmission protocols are but a few examples.

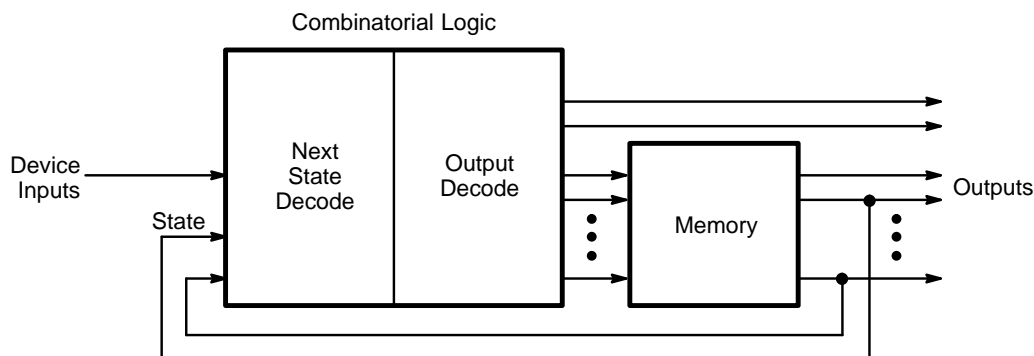
Typically, the details of control logic are the last to be settled in the design cycle, since they are continuously affected by changing system requirements and feature enhancements. Programmable logic is a forgiving solution for control logic design because it allows easy modifications to be made without disturbing PC board layout. Its flexibility provides an escape valve that permits design changes without impacting time-to-market.

A majority of registered PAL device applications are sequential control designs where state machine design techniques are employed. As technology advances, new high-speed and high-functionality devices are being introduced which simplify the task of state machine design. A broad range of different functionality-and-performance solutions are available for state machine design. In this discussion we will examine the functions performed by state machines, their implementation on various devices, and their selection.

What Is a State Machine?

A state machine is a digital device that traverses through a predetermined sequence of states in an orderly fashion. A state is a set of values measured at different parts of the circuit. A simple state machine can consist of PAL-device based combinatorial logic, output registers, and buried (state) registers. The state in such a sequencer is determined by the values stored in the buried and/or output registers.

A general form of a state machine can be depicted as a device shown in Figure 1. In addition to the device inputs and outputs, a state machine consists of two essential elements: combinatorial logic and memory (registers). This is similar to the registered counter designs discussed previously, which are essentially simple state machines. The memory is used to store the state of the machine. The combinatorial logic can be viewed as two distinct functional blocks: the next state decoder and the output decoder (Figure 2). The next state decoder determines the next state of the state machine while the output decoder generates the actual outputs. Although they perform two distinct functions, these are usually combined into one combinatorial logic array as in Figure 1.



90005A-1

Figure 1. Block Diagram of a Simple State Machine

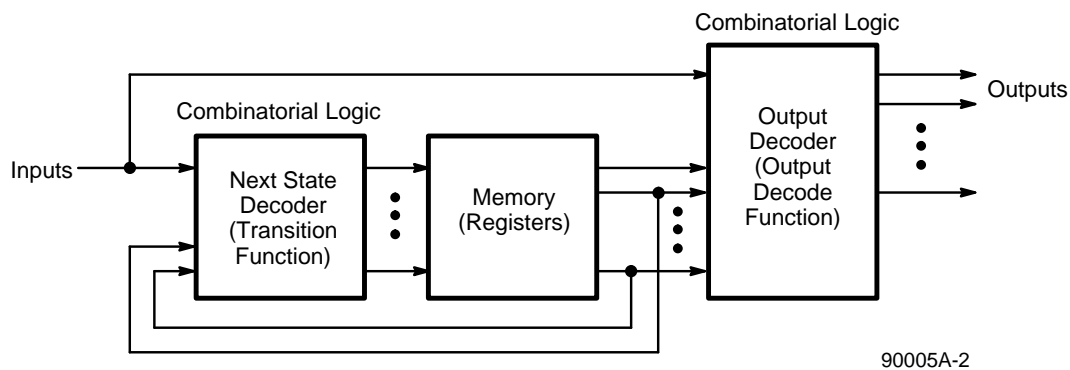


Figure 2. State Machine, with Separate Output and Next State Decoders

The basic operation of a state machine is twofold:

1. It traverses through a sequence of states, where the next state is determined by next state decoder, depending upon the present state and input conditions.
2. It provides sequences of output signals based upon state transitions. The outputs are generated by the output decoder based upon present state and input conditions.

Using input signals for deciding the next state is also known as branching. In addition to branching, complex sequencers provide the capability of repeating sequences (looping) and subroutines. The transitions from one state to another are called *control sequencing* and the logic required for deciding the next states is called the *transition function* (Figure 2).

The use of input signals in the decision-making process for *output generation* determines the type of a state machine. There are two widely known types of state machines: Mealy and Moore (Figure 3). Moore state machine outputs are a function of the present state only. In the more general Mealy-type state machines, the outputs are functions of both the state and the input signals. The logic required is known as the *output function*. For either type, the control sequencing depends upon both states and input signals.

Most practical state machines are synchronous sequential circuits that rely on clock signals to trigger the state transitions. A single clock is connected to all of the state and output edge-triggered flip-flops, which allows a state change to occur on the rising edge of the clock. Asynchronous state machines are also possible, which utilize the propagation delay in combinatorial logic for the memory function of the state machine. Such machines are highly susceptible to hazards, hard to design and are seldom used. In our discussion we will focus solely on sequential state machines.

State Machine Applications

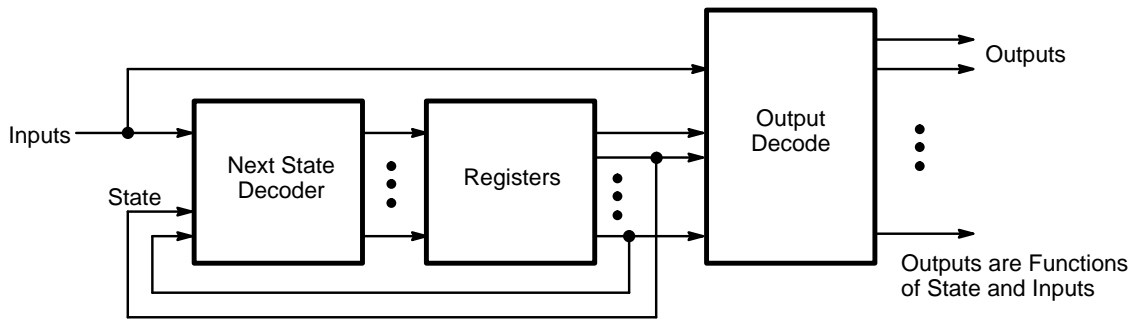
State machines are used in a number of system control applications. A sampling of a few of the applications, and how state machines are applied, is described below.

As sequencers for digital signal processing (DSP) applications, state machines offer speed and sufficient functionality without the overkill of complex microprocessors. For simple algorithms, such as those involved in performing a Fast Fourier Transform (FFT), a state machine can control the set of vectors that are multiplied and added in the process. For complex DSP operations, a programmable DSP may be better. On the other hand, the programmable DSP solution is not likely to be as fast as the dedicated hardware approach.

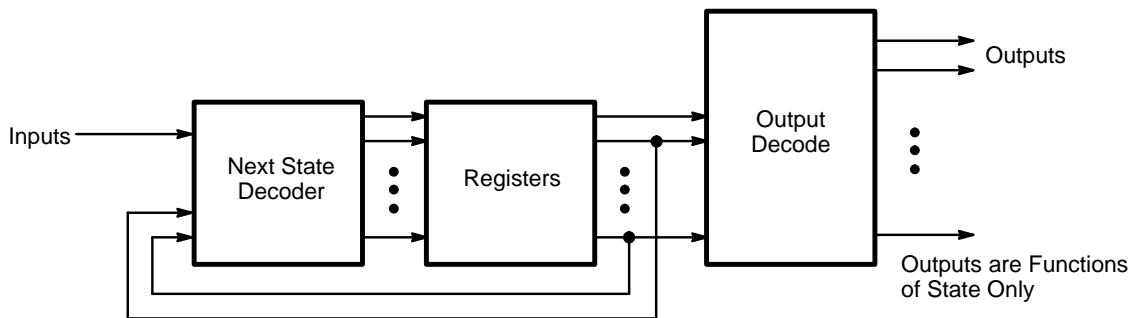
Consider the case of a video controller. It generates addresses for scanning purposes, using counters with various sequences and lengths. Instead of implementing these as actual counters, the sequences involved can be “unlocked” and implemented, instead, as state machine transitions. There is an advantage beyond mere economy of parts. A count can be set or initiated, then left to take care of itself, freeing the microprocessor for other operations.

In peripheral control the simple state machine approach can be very efficient. Consider the case of run-length-limited (RLL) code. Both encoding and decoding can be translated into state machines, which examine the serial data stream as it is read, and generate the output data.

Industrial control and robotics offer further areas where simple control functions are required. Such tasks as mechanical positioning of a robot arm, simple decision making, and calculation of a trigonometric function, usually does not require the high-power solution of microprocessors with stacks and pointers. Rather, what is required is a device that is capable of storing a limited number of states and allows simple branching upon conditions.



a. Mealy State Machines



b. Moore State Machines

90005A-3

Figure 3. The Two Standard State Machine Models

Data encryption and decryption present similar problems to those encountered in encoding and decoding for mass media, only here it is desirable to make the scheme not so obvious. A programmable state machine device with a security Bit is ideal for this because memory is internally programmed and cannot be accessed by someone tampering with the system.

Functions Performed

All the system design functions performed by controllers can be categorized as one of the following state machine functions:

- Arbitration
- Event monitoring
- Multiple condition testing
- Timing delays
- Control signal generation

Later we will take a design example and illustrate how these functions can be used when designing a state machine.

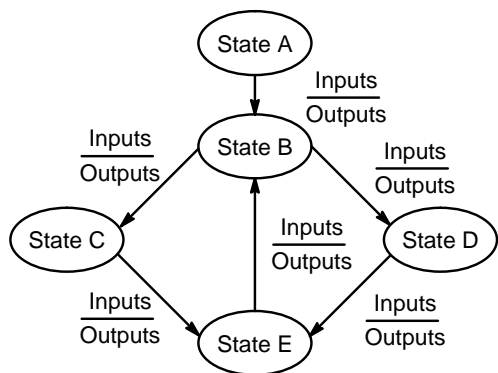
State Machine Theory

Let us take a brief look at the underlying theory for all sequential logic systems, the *finite state machine* (FSM), or simply state machine.

Those parts of digital systems whose outputs depend on their past inputs as well as their current ones can be modeled as finite state machines. The “history” of the machine is summed up in the value of its internal state. When a new input is presented to the FSM, an output is generated which depends on this input and the present state of the FSM, and the machine is caused to move into new state, referred to as the next state. This new state also depends on both the input and present state. The structure of an FSM is shown pictorially in Figure 2. The internal state is stored in a block labeled “memory.” As discussed earlier, two combinatorial functions are required: the transition function, which generates the value of the next state, and the output function, which generates the state machine output.

State Diagram Representation

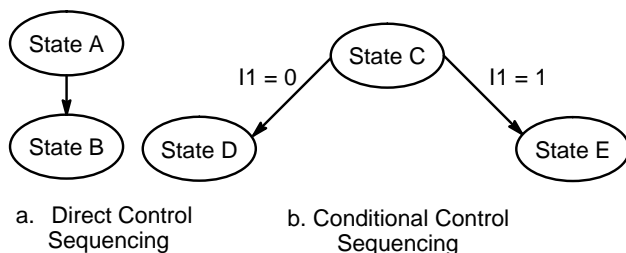
The behavior of an FSM may be specified in graphical form as shown in Figure 4. This is called a state diagram, or state transition diagram. Each bubble represents a state, and each arrow represents a transition between states. Inputs that cause the transitions are shown next to each transition arrow.



90005A-4

Figure 4. State Machine Representation

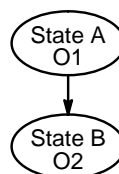
Control sequencing is represented in the state transition diagram as shown in Figure 5. Direct control sequencing requires an unconditional transition from state A to state B. Similarly conditional control sequencing shows a conditional transition from state C to either state D or state E, depending upon input signal I1.



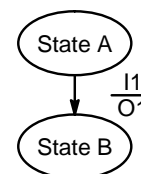
90005A-5

Figure 5. Control Sequencing

For Moore machines the output generation is represented by assigning outputs with states (bubbles) as shown in Figure 6. Similarly, for Mealy machines conditional output generation is represented by assigning outputs to transitions (arrows), as was shown in Figure 4. More detail on Mealy and Moore output generation is given later.



a. Moore Machine



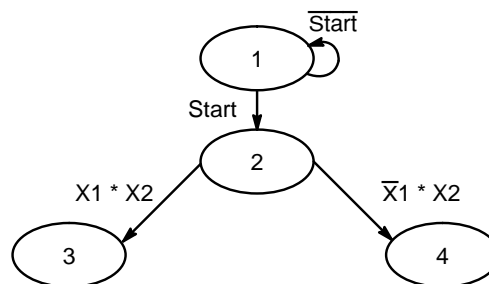
b. Mealy Machine

90005A-6

Figure 6. Output Generation

For this notation, there is a specification uncertainty as to which signals are outputs or inputs, as they both occur on the drawing next to the arrow in which they are active. This is usually resolved by separating the input and output signals names with a line (Figures 4 and 6). Sometimes an auxiliary pin list detailing the logic polarity and input or output designations is also used.

State transition diagrams can be made more compact by writing on the transitions not the input values that cause the transition, as in Figure 4, but a Boolean expression defining the input combination or combinations that cause this transition. For example, in Figure 7, some transitions have been shown for a machine with inputs START, X1, and X2. In the transition between states 1 and 2, the inputs X1 and X2 are ignored (that is, they are "don't cares") and thus do not appear on the diagram. This saves space and makes the function more obvious.



90005A-7

Figure 7. State Transition Diagram with Mnemonics

There can be a problem with this method if one is careless. The state transitions in figure 8 show what can happen. There are three input combinations, (I0, I1, I2, I3) = {1011}, {1101} and {1111}, which make both (I0 * I2 + I3) and (I0 * I1 + I0 * I2) true. Since a transition to two next states is impossible, this is an error in the

specification. It must either be guaranteed that these input combinations never occur, or the transition conditions must be modified. In this example, changing $(I0 * I1 + I0 * I2)$ to $(I0 * I1 + I0 * I2) * /I3$ would solve the problem.

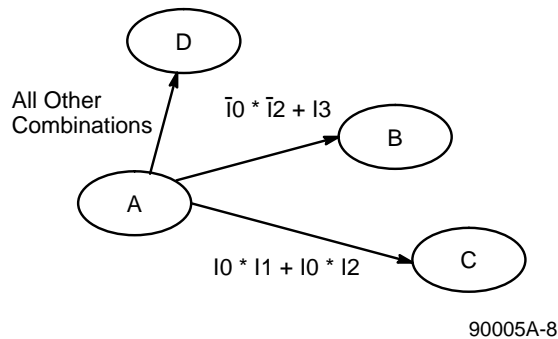


Figure 8. State Diagram with Conflicting Branch Conditions

State Transition Table Representation

A second method for state machine representation is the tabular form known as the state transition table, which has the format shown in Table 1. Listed along the top are all the possible input bit combinations and internal states. Each row gives the next state and the next output; thus, the table specifies the transition and output functions. However, this type of table is not suitable for specifying practical machines in which there is a large number of inputs, since each input combination defines a row of the table. For example, with 10 inputs, 1024 rows would be required!

Table 1. A State Transition Table

Present State	Inputs	Next State	Outputs Generated
S0 – Sn	I0 – Im	S0 – Sn	O0 – Op

Flowcharts

Another popular notation is based on flowcharts. In this notation, states are represented by rectangular boxes, and alternative state transitions are determined by strings of diamond-shaped boxes. The elements may have multiple entry points, but in general have only one exit. The state name is written as the first entry in the rectangular state box. Any Moore outputs present are written next in the state box, with a caret (^) following those that are unregistered. The state code assignment, if it is known, is written next to the upper right corner of the state box. Decision boxes are diamond or hexagonal shaped boxes containing either an input signal or a logic expression. Two exits labeled “0” and “1” lead to either another decision box, a state box, or a Mealy output.

The rounded oval is used for Mealy machine outputs. Again, a caret follows those outputs that are unregistered. All the boxes may need to be expanded to accommodate a number of output signals or a larger expression.

The use of these symbols is shown in Figure 9. Each path, through the decision boxes from one state to another defines a particular combination or set of combinations of the input variables. A path does not have to include all input variables; thus, it accommodates “don’t cares.” These decision trees take more space than the expressions would, but in many practical cases, state machine controllers only test a small subset of the input variables in each state and the trees are quite manageable. Also, the chain of decisions often mirrors the designer’s way of thinking about the actions of the controller. It is important to note that these tests are not performed sequentially in the FSM; all are performed in parallel by the FSM’s state transition logic.

A benefit of this method of specifying transitions is that the problem of Figure 8 can be avoided. Such a conflict would be impossible as one path cannot diverge to define paths to two states.

This flowchart notation can be compacted by allowing more complex decisions, when there is no danger of conflicts due to multiple next states being defined. Expressions can be tested, as shown in Figure 10a, or multiple branches can extend from a decoding box, as in Figure 10b. In the second case, it is convenient to group the set of binary inputs into a vector, and branch on different values of this vector.

The three methods of state machine representation state diagrams, state tables, and flowcharts

are all equivalent and interchangeable, since they all describe the same hardware structure. Each style has its own particular advantages. Although most popular, the state transition diagrams are more complex for problems where state transitions depend on many inputs, since the transition conditions are written directly on the transition arrows. Although cumbersome, the state tables allow the designer tight control over signal logic. Flowcharts are convenient for small problems where there are not more than about ten states and where up to two or three inputs or input expressions are tested in each state. For larger problems, they can become ungainly.

Once a state machine is defined, it must be implemented on a device. Software packages are then used to implement the design on a device. The task is to convert the state machine description into transition and output functions. Software packages also account for device-specific architectural variations and limitations, to provide a uniform user interface.

Some software packages accept all three different state machine representations directly as design inputs. However, the most prevalent design methodology is to convert the three state machine design representations to a simple textual representation. Textual representations are accepted by most software packages although the syntax varies.

Since the most common of all state machine representations is the state transition diagram representation, we will use it in all subsequent discussions. Transition table and flowchart representation implementations will be very similar.

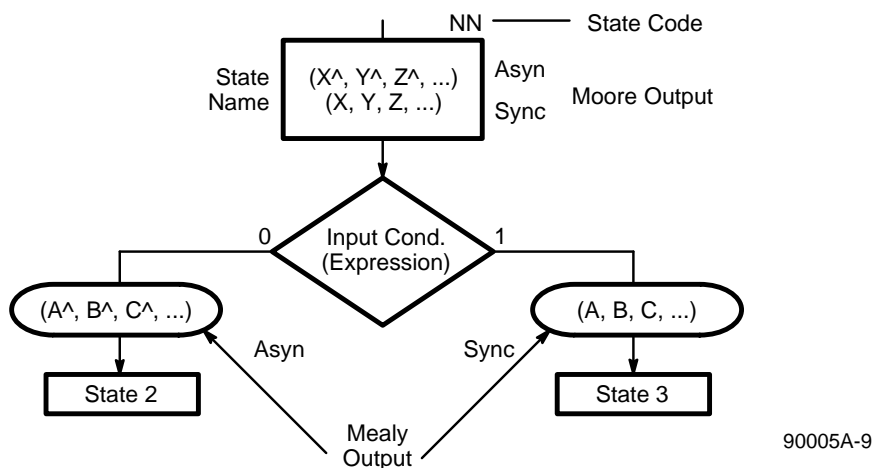


Figure 9. Flowchart Notation

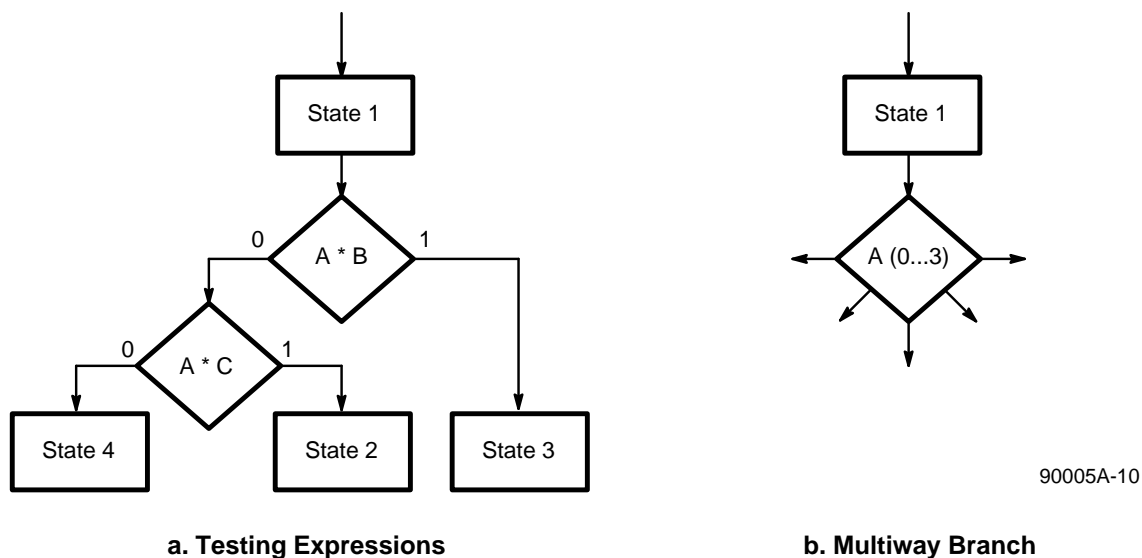


Figure 10. Using Flowcharts

State Machine Types: Mealy & Moore

With the state machine representation clarified, we can now return to the generic sequencer model of Figure 1, which has been labeled (Figure 11) to show the present state (PS), next state (NS), and output (OB, OA). This will illustrate how Mealy and Moore machines are implemented with most sequencer devices that provide a single combinatorial logic array for both next state and output decode functions. There are four ways of using the sequencer, two of which implement Moore machines and two Mealy. First, let us look at the Mealy forms.

The standard Mealy form is shown in Figure 12, where the signals are labeled as in Figure 11 to indicate which registers and outputs are used. The register outputs PS are fed back into the array and define the present state. The combinatorial logic implements the transition function, which produces the next state flip-flop inputs NS, and the output function, which produces the machine output OB. This is the asynchronous Mealy form.

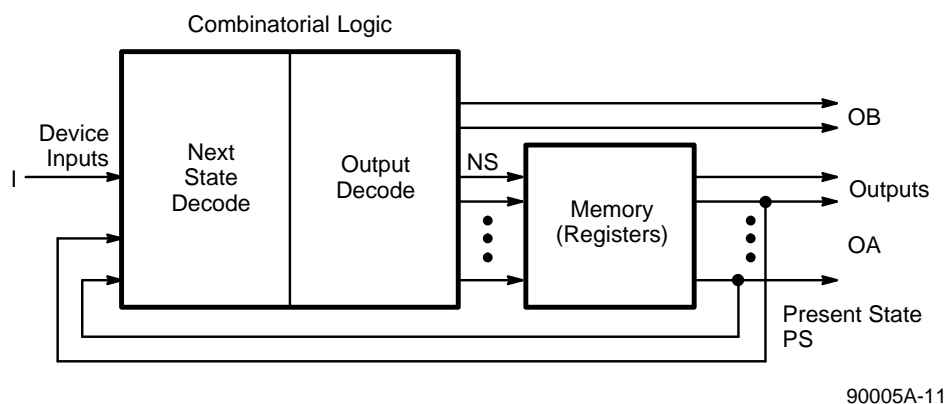


Figure 11. Generic Model of an FSM

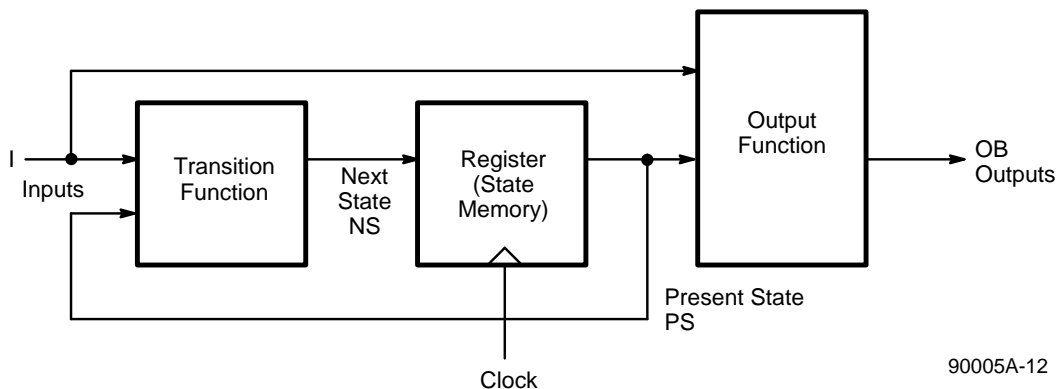
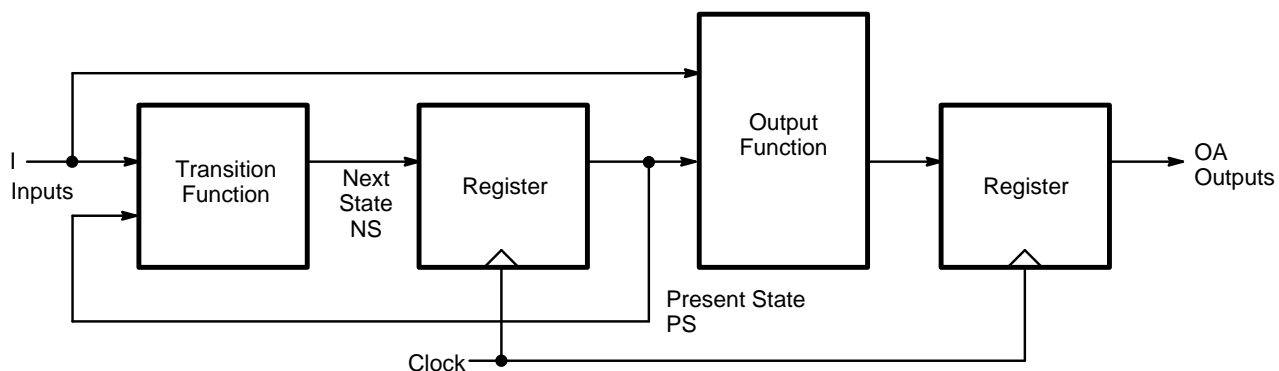


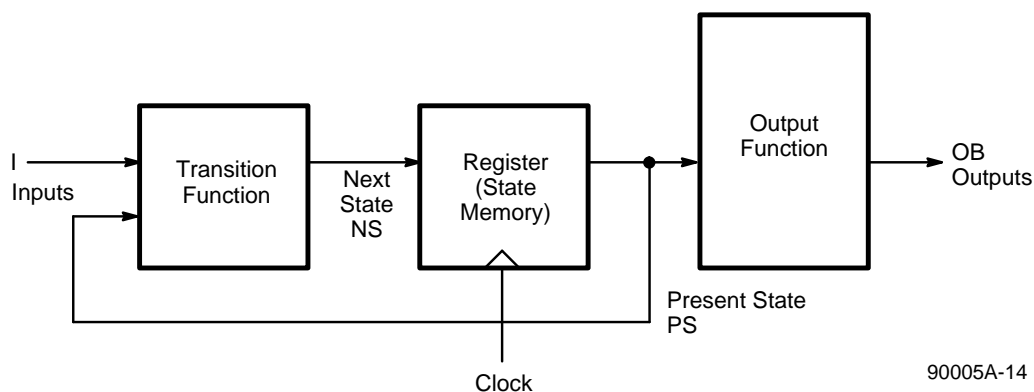
Figure 12. Asynchronous Mealy Form

An alternative Mealy form is shown in Figure 13. Here the outputs are passed through an extra output register (OA) and thus, do not respond immediately to input changes. This is the synchronous Mealy form.



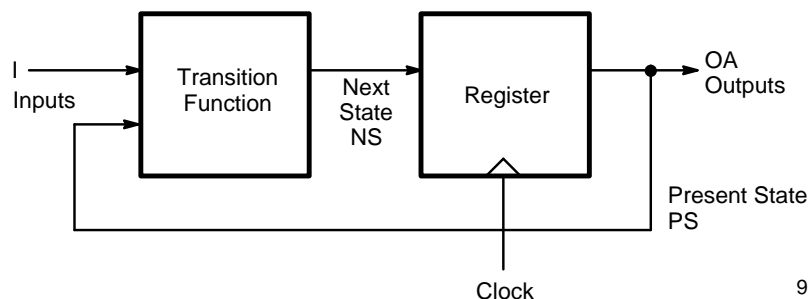
90005A-13

Figure 13. Synchronous Mealy Form



90005A-14

Figure 14. Asynchronous Moore Form



90005A-15

Figure 15. Synchronous Moore Form

The standard Moore form is given in Figure 14. Here the outputs OB depend only on the present state PS. This is the asynchronous Moore form. The synchronous Moore form is shown in Figure 15. In this case the combinational logic can be assumed to be the unity function. The outputs (OB) can be generated directly along with the

present state (PS). Although these forms have been described separately, a single sequencer is able to realize a machine that combines them, provided that the required paths exist in the device.

In the synchronous Moore form, the outputs occur in the state in which they are named in the state transition diagram. Similarly, in the asynchronous Mealy and Moore forms the outputs occur in the state in which they are named, although delayed a little by the propagation delay of the output decoder. This is because they are combinatorial functions of the state (and inputs in the Mealy case).

However, the synchronous Mealy machine is different. Here an output does not appear in the state in which it is named, since it goes into another register first. It appears when the machine is in the next state, and is thus delayed by one clock cycle. The state diagram in Figure 16 illustrates all the possibilities on a state transition diagram.

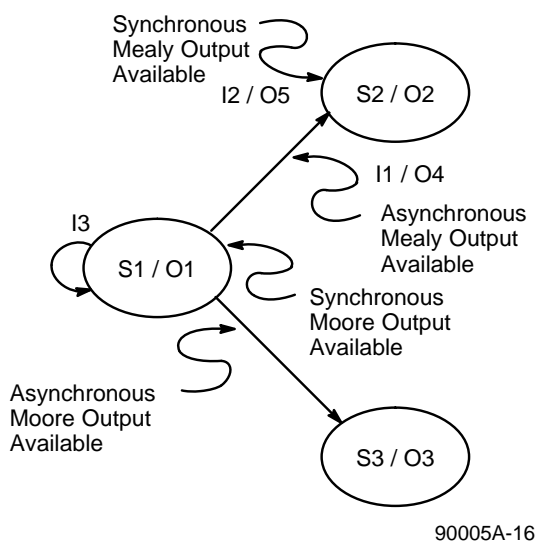


Figure 16. State Diagram Labelling for Different Output Types

As a matter of notation, Moore outputs are often placed within the state bubble and Mealy outputs are placed next to the path or arrow that activates them.

The relationship of Mealy and Moore, synchronous and asynchronous outputs to the states is shown in Figure 17.

Device Selection Considerations

There are three major criteria for selecting the correct state machine device for a design:

- Number of inputs/outputs
 - I/O flexibility
 - Number of output registers
- Speed
- Intelligence/functionality
 - Number of product terms
 - Type of flip-flops
 - Number of state registers

Number of I/Os

The number of inputs, outputs and I/O pins determine the signals that can be sampled or generated by a state machine.

Timing and Speed

The timing considerations for sequencer design are similar to those for registered logic design. A system clock cycle forms the basic kernel for evaluating control function behavior. For the most part, all input and output functions are specified in relationship to the positive edge. Registered outputs are available after a period of time t_{CO} , the clock-to-output propagation delay. Asynchronous outputs require an additional propagation delay (t_{PD}) before they are valid.

For the circuit to operate reliably, all the flip-flop inputs must be stable at the flip-flop by the minimum set-up time (t_S) of the flip-flops before the next active clock edge. If one of the inputs changes after this threshold, then the next state or synchronous output could be stored incorrectly; the circuit may even malfunction. To avoid this, the clock period (t_P) must be greater than the sum of the set-up time of the flip-flops and the clock to output time ($t_S + t_{CO}$). This determines the minimum clock period and hence the maximum clock frequency, f_{MAX} , of the circuit. Metastability and erroneous system operation may occur if these specifications are violated.

The timing relationships are shown in Figure 18. In each cycle there are two regions: the stable region, when all signals are steady, and the transition region, when the machine is changing state and signals are unstable. The active clock edge causes the flip-flops to load the value of the new state that has been set up at their inputs.

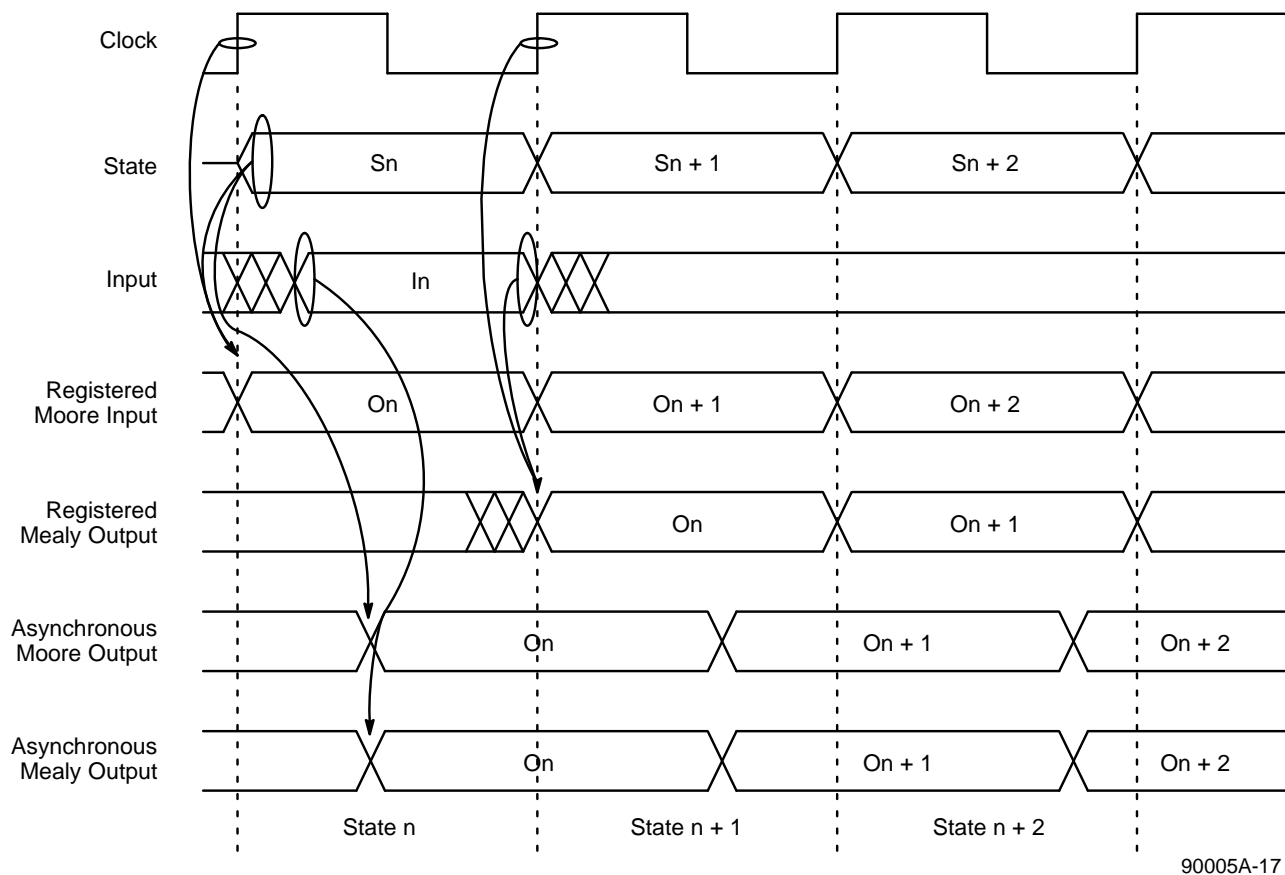


Figure 17. State Machine Timing Diagram

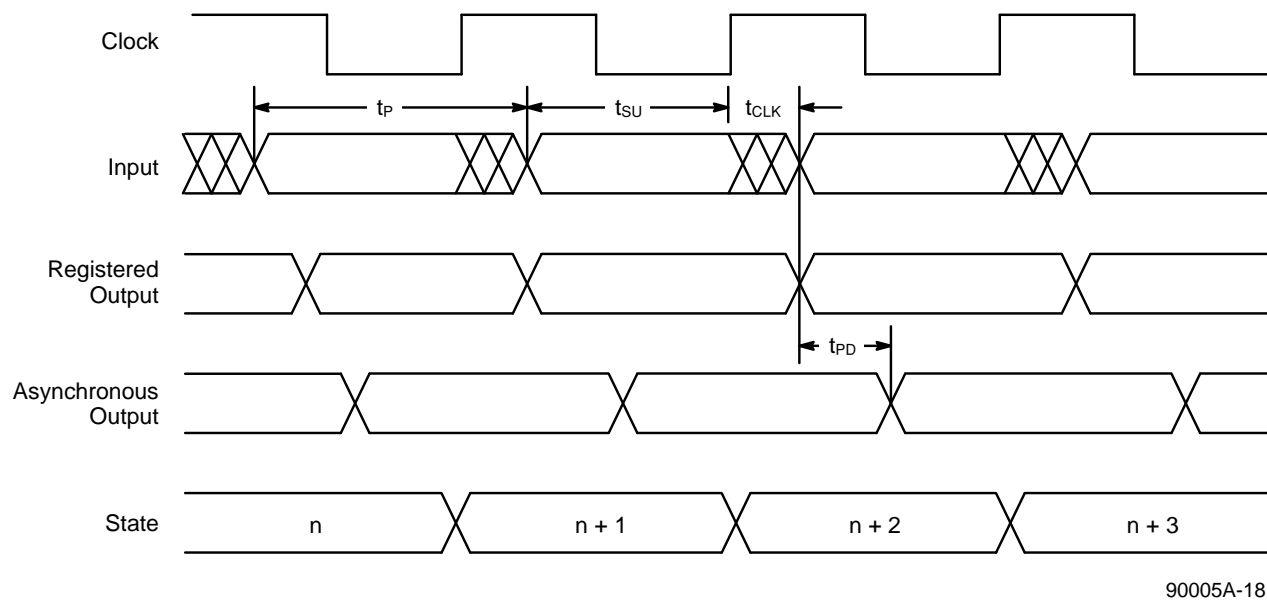


Figure 18. Timing Diagram for Maximum Operating Frequency

At a time after this, the present state and output flip-flop outputs will start to change to their new values. After a time has elapsed, the slowest flip-flop output will be stable at its new value. Ignoring input changes for the moment, the changes in the state register cause the combinatorial logic to start generating new values for the asynchronous outputs and the inputs to the flip-flops. If the propagation delay of the logic is t_{PD} , then the stable period will start at a time equal to the sum of the maximum values of t_{CO} , and t_{PD} .

Asynchronous Inputs

The timing of the inputs to an asynchronous state machine is often beyond the control of the designer and may be random, such as sensor or keyboard inputs, or they may come from another synchronous system that has an unrelated clock. In either case no assumptions can be made about the times when inputs can or cannot

arrive. This fact causes reliability problems that cannot be completely eliminated, but only reduced to acceptable levels.

Figure 19 shows two possible transitions from state “S1” (code 00) either back to itself, or to state “S2” (code 11). Which transition is taken depends on input variable “A” which is asynchronous to the clock. The transition function logic for both state bits B1 and B2 include this input. The input A can appear in any part of the clock cycle. For the flip-flops to function correctly, the logic for B1 and B2 must stabilize correctly before the clock. The input should be stable in a window t_s (setup time) before the clock and t_h (hold time) after the clock. If the input changes within this window, both the flip-flops may not switch, causing the sequence to jump to states 01 or 10, which are both undefined transitions. This type of erroneous behavior is called an input race.

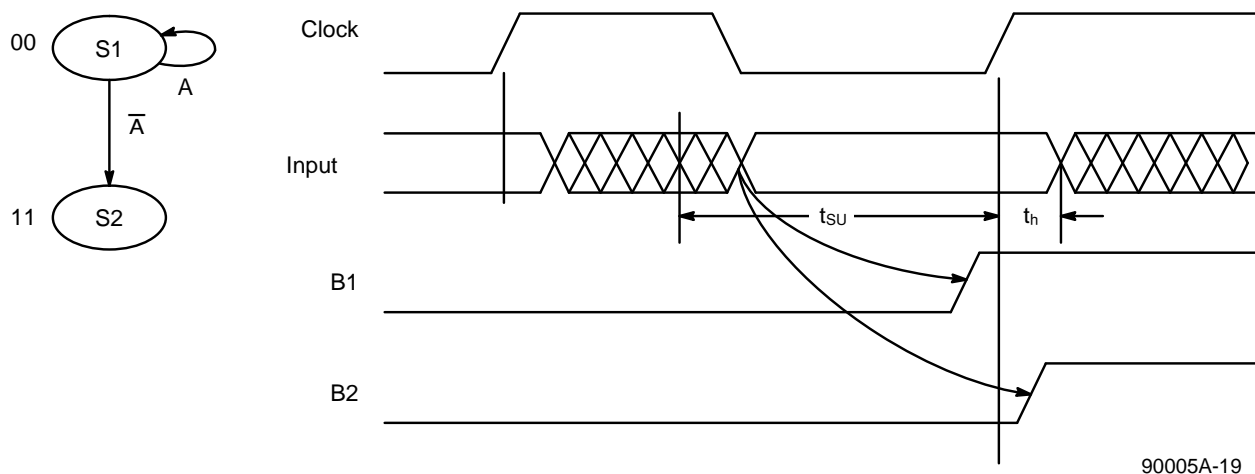


Figure 19. Asynchronous Input Cascading Race

A solution to this problem is to change the state assignment so that only one state variable depends on the asynchronous input. Thus, the 11 code must be changed to 01 or 10. Now, with only one unsynchronized flip-flop input, either the input occurs in time to cause the transition, or it does not, in which case no transition occurs. In the case of a late input, the machine will respond to it one cycle later, provided that the input is of sufficient duration.

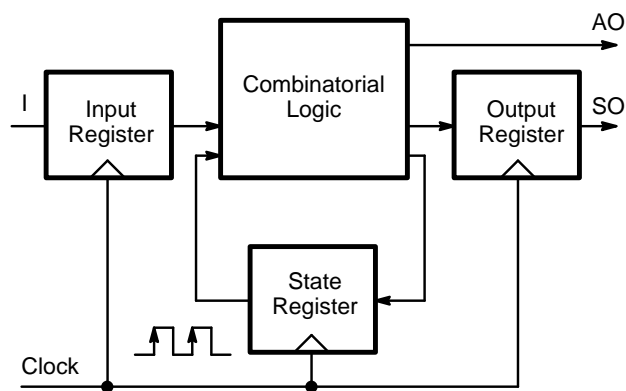
There is still the possibility of an input change violating the setup time of the internal flip-flop, driving it into a metastable state. This can produce system failures that can be minimized, but never eliminated. The same problem arises when outputs depend on an asynchronous input.

Very little can be done to handle asynchronous inputs without severely constraining the design of the state machine. The only way to have complete freedom in the use of inputs is to convert them into synchronous inputs. This can be done by allocating a flip-flop to each input as shown in Figure 20. These synchronizing flip-flops are clocked by the sequencer clock, and may even be the sequencer's own internal flip-flops. This method is not foolproof, but significantly reduces the chance of metastability occurring.

Functionality

The functionality of different devices is difficult to compare since different device architectures are available. The number of registers in a device determines the number of state combinations possible. However, all the possible state combinations are not necessarily usable, since other device constraints may be reached. The number of registers does give an idea of the functionality achievable in a device. Other functionality measures include the number of product terms and type of flip-flop. One device may be stronger than another in one of these measures, but overall may be less useful due to other shortcomings. Choosing the best device involves both skill and experience.

In order to give an idea of device functionality, we will consider each of the architecture options available to the designer and evaluate its functionality.



90005A-20

Figure 20. Input Synchronizing Register

PAL Devices as Sequencers

A vast majority of state machine designs are implemented with PAL devices. Early versions of software required the user to manually write the sum-of-products Boolean equations for using PAL devices. Second generation software allows one to specify the design in "state machine syntax," and handles the translation to sum-of-products logic automatically. PAL devices implement the output and transition functions in sum-of-products form through a user-programmable AND array and a fixed OR array.

PAL devices deliver the fastest speed of any sequencer and are ideally suited for simple control applications characterized by few input and output signals interacting within a dedicated controller in a sequential manner. The number of flip-flops in a typical PAL device range from 8 to 12, which offer potentially more than one thousand state values. Since some of the flip-flops are used for outputs, and the number of product terms is limited, the usable number of states is reduced drastically. Generally, up to about 35 states can be utilized.

PAL Device Flip-Flops

PAL device based sequencers implement small state machine designs, which have a relatively large number of output transitions. Since the output registers change with most state transitions, they can be used simultaneously as state registers, once the state values are carefully selected. Most PAL devices are used for small state machines, and efficiently share the same register for output and state functions. High-functionality PAL device based sequencers provide dedicated buried state registers when sharing is difficult.

As a state machine traverses from one state to another, every output either makes a transition (changes logic level) or holds (stays at the same logic level). Small state machine designs require relatively more transitions and fewer holds. As designs get larger, state machines statistically require relatively fewer transitions and more holds.

Most PAL devices provide D-type output registers. D-type flip-flops use up product terms only for active transitions from logic LOW to HIGH level, and for holds for logic HIGH level only. J-K, S-R, and T-type flip-flops use up product terms for both LOW-to-HIGH and HIGH-to-LOW transitions, but eliminate hold terms. Thus, D-type flip-flops are more efficient for small state machine designs. Some PAL devices offer the capability of configuring the flip-flops as J-K, S-R or T-types, which are more efficient for large state machine designs since they require no hold terms.

Many examples of PAL-device-based sequencers can be found in system time base functions, special counters, interrupt controllers, and certain types of video display hardware.

PAL devices are produced in a variety of technologies for multiple applications, and provide a broad range of speed-power options.