15 STATE MACHINES

STATE MACHINE TYPES

There are two types of state machines: Mealy machines and Moore machines. You can model both types of machines in Verilog. The difference between Mealy and Moore machines is in how outputs are generated. In a Moore machine, the outputs are a function of the current state. This implies that the outputs from the Moore machine are synchronous to the state changes. In a Mealy machine, the outputs are a function of both the state and the inputs.

A state machine can be broken down into three parts: The state register, the next-state logic, and the output logic.

A state machine can be depicted as shown in Figure 15-1.

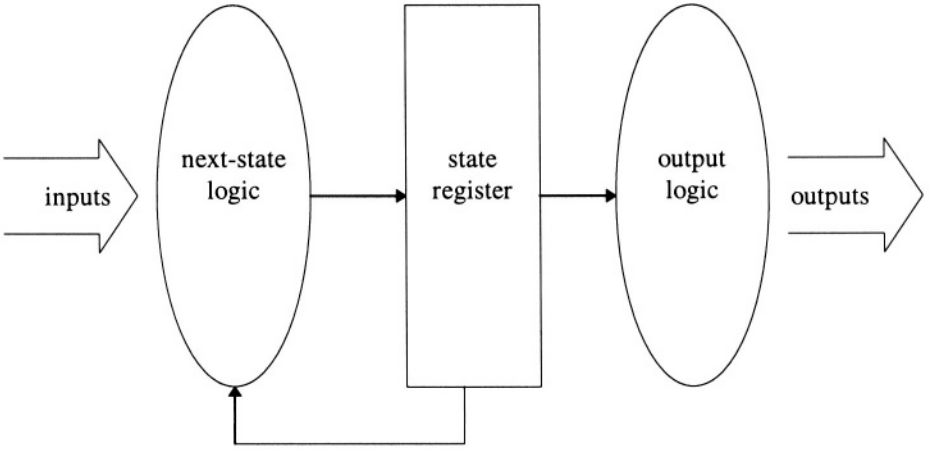

Figure 15-1 Moore State Machine

Figure 15-1 can be modified to bring the inputs through to the output logic, thus creating a Mealy state machine, as shown in Figure 15-2.

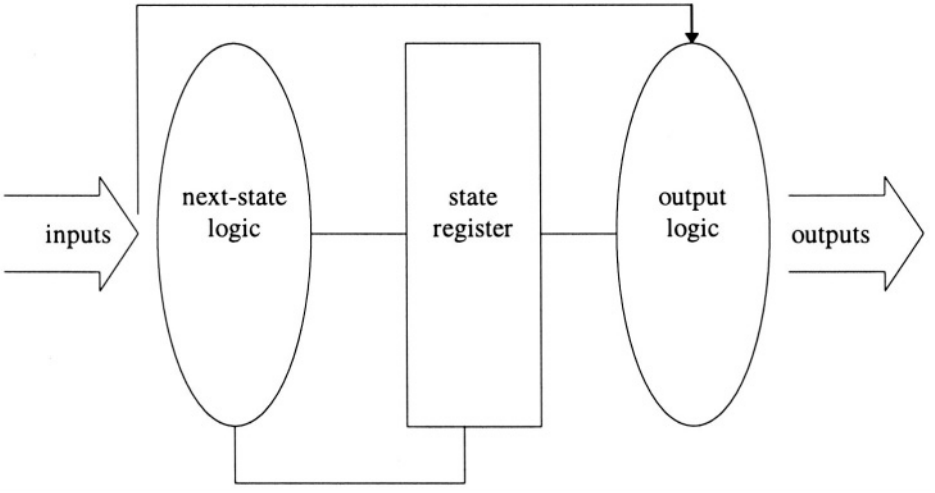

Figure 15-2 Mealy State Machine

To model a state machine in Verilog, you must model each of the three parts of the state machine.

STATE MACHINE MODELING STYLE

Because a state machine is made up of three parts, you have a choice whether to model each section independently, or to try to combine the parts into one section of the model.

Table 15-1 shows some interesting combinations:

Table 15-1 State Machine Styles

Style	State Register	Next-State Logic	Output Logic
1	Separate	Separate	Separate
2	Combined	Combined	Separate
3	Separate	Combined	Combined
4	Combined	Combined	Combined
5	Combined	Separate	Combined

In the first style, each of the functional blocks is modeled in a separate *always* block. The state register logic is modeled as an *always* block, and the next-state logic and output logic are separate *always* blocks, representing combinatorial logic. Because the output section can be modeled as either sensitive only to changes on state or also sensitive to changes on inputs, you can use this to model both Mealy and Moore machines. This style is the most modular; it may take a few more lines of Verilog code, though it may be the easiest to maintain.

The second style combines the next-state logic and the state register. This style is a good style to use because the next-state logic and state register are strongly related. This style is more compact than the first, and may even be more efficient because the next-state logic is only evaluated on clock edges, rather than whenever an input changes. If your state machine has many inputs that change frequently, this may be a better style to use than the first. This style has the output as a separate section so you can use this style to model both Mealy and Moore machines.

The third style leaves the state register in a separate *always* block, while combining the next-state logic and the output logic. Because the next-state logic and the output logic may be combinatorial, combining them still allows for modeling both Mealy and Moore machines. However, this grouping does not tend to help the readability of your code; styles 1 and 2 are easier to model and maintain.

The fourth style combines everything into one *always* block. The *always* block is sensitive only to the clock for the state register, so this implies the outputs only change with the state. Thus this style will only create Moore machines.

The fifth and final style combines the state register and output logic into one *always* block, so again, this only creates Moore machines.

To demonstrate all these styles, with Moore and Mealy variations on them, we will use a simple example. This example won't be the traffic light controller, vending machine, or a completely trivial state machine, but instead an automatic food cooker. This cooker has a supply of food that it can load into its heater when requested. The cooker then unloads the food when the cooking is done.

Besides *clock,* the inputs to this state machine are *start* (which starts a load/cook/unload cycle); *temp_ok* (a temperature sensor that detects when the heater is done preheating); *done* (a signal from a timer or sensor that detects when the cooking cycle is complete); and *quiet* (a final input that selects if the cooker should beep when the food is ready).

The outputs from the machine are *load* (a signal that sends food into the cooker); *heat* (a signal that turns on the heating element, which has its a built-in temperature control); *unload* (a signal that removes the food from the cooker and presents it to the diner); and *beep* (a signal that alerts the diner when the food is done).

Example 15-1 Style 1 Moore State Machine

```
module auto_oven_style_1_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE     'b000
`define PREHEAT  'b001
`define LOAD     'b010
`define COOK     'b011
`define EMPTY    'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;
// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
     `IDLE: if (start) next_state=`PREHEAT;
     `PREHEAT: if(temp_ok) next_state = `LOAD;
     `LOAD: next_state = `COOK;
     `COOK: if (done) next_state=`EMPTY;
     `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase
end

// Output logic
always @(state) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

In style 1, as shown in Example 15-1, each section of the state machine is modeled with a separate *always* block. Style 1 can be used to represent either a Moore machine or a Mealy machine for our automatic oven. The difference between the two styles is seen in the different behavior of the *quiet* input and *beep* output. With the Moore machine the diner must wait through the entire EMPTY state for the beeper to be quiet. The Mealy version of this is for those diners who want the beeper to sound, and then jump up to turn it off, as shown in Example 15-2.

Example 15-2 Style 1 Mealy State Machine

```
module auto_oven_style_1_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE     'b000
`define PREHEAT  'b001
`define LOAD     'b010
`define COOK     'b011
`define EMPTY    'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;
// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
     `IDLE: if (start) next_state=`PREHEAT;
     `PREHEAT: if(temp_ok) next_state = `LOAD;
     `LOAD: next_state = `COOK;
     `COOK: if (done) next_state=`EMPTY;
     `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase
end

// Output logic
always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Style 2 can also be used to express both Moore and Mealy machines. Example 15-3 and Example 15-4 are just slightly shorter because the next-state logic and register are combined into one block of code. This also eliminates the need for the *next_state* temporary variable.

Example 15-3 Style 2 Moore Machine

```
module auto_oven_style_2_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
```

```
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock)begin
  case (state)
    `IDLE: if (start) state=`PREHEAT;
    `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
    `LOAD: state <= #(`REG_DELAY) `COOK;
    `COOK: if (done) state=`EMPTY;
    `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
end
// Output logic
always @(state) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Example 15-4 Style 2 Mealy Machine

```
module auto_oven_style_2_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock)begin
  case (state)
    `IDLE: if (start) state=`PREHEAT;
    `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
    `LOAD: state <= #(`REG_DELAY) `COOK;
    `COOK: if (done) state=`EMPTY;
```

```
    `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
end
// Output logic
always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0 ;
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD |
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Style 3 combines the next-state logic and output logic. In Example 15-5, this modeling will result in a Mealy machine. It is possible to use this style to describe a Moore machine.

Example 15-5 Style 3 Mealy Machine

```
module auto_oven_style_3_mealy(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock)
  state <= #(`REG_DELAY) next_state;

// next state logic
always @(state or start or temp_ok or done or quiet) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase

//output logic
  if(state == `LOAD) load = 1; else load = 0;
```

```
  if(state == `EMPTY) unload =1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep =1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end

endmodule
```

Style 4 combines everything into one big block, which yields a Moore machine.

Example 15-6 Style 4 Moore Machine

```
module auto_oven_style_4_moore(clock, start, temp_ok, done,
  quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock)begin
  case (state)
     `IDLE: if (start) state=`PREHEAT;
     `PREHEAT: if(temp_ok) state <= #(`REG_DELAY) `LOAD;
     `LOAD: state <= #(`REG_DELAY) `COOK;
     `COOK: if (done) state=`EMPTY;
     `EMPTY: state <= #(`REG_DELAY) `IDLE;
    default: state <= #(`REG_DELAY) `IDLE;
  endcase
  if(state == `LOAD) load <= #(`REG_DELAY) 1;
    else load <= #(`REG_DELAY) 0;
  if(state == `EMPTY) unload <= #(`REG_DELAY) 1;
    else unload <= #(`REG_DELAY) 0;
  if(state == `EMPTY && quiet == 0) beep <= #(`REG_DELAY) 1;
    else beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat <= #(`REG_DELAY) 1;
     else heat <= #(`REG_DELAY) 0;
end
endmodule
```

Style 5 combines the state register and output sections, and results in either a Moore machine or registered outputs.

Example 15-7 Style 5 Moore Machine

```
module auto_oven_style_5_moore(clock, start, temp_ok, done,
quiet, load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
reg [2:0] state, next_state;
`define IDLE    'b000
`define PREHEAT 'b001
`define LOAD    'b010
`define COOK    'b011
`define EMPTY   'b100

// State register block
always @(posedge clock) begin
  state <= #(`REG_DELAY) next_state;
// Output logic
  if(state == `LOAD) load <= #(`REG_DELAY) 1;
    else load <= #(`REG_DELAY) 0;
  if(state == `EMPTY) unload <= #(`REG_DELAY) 1;
    else unload <= #(`REG_DELAY) 0;
  if(state == `EMPTY && quiet == 0) beep <= #(`REG_DELAY) 1;
    else beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat <= #(`REG_DELAY) 1;
    else heat <= #(`REG_DELAY) 0;

end

// next state logic
always @(state or start or temp_ok or done) begin
  next_state = state; // default to stay in current state
  case (state)
    `IDLE: if (start) next_state=`PREHEAT;
    `PREHEAT: if(temp_ok) next_state = `LOAD;
    `LOAD: next_state = `COOK;
    `COOK: if (done) next_state=`EMPTY;
    `EMPTY: next_state = `IDLE;
    default: next_state = `IDLE;
  endcase
end

endmodule
```

With all of these styles to choose from, which one is best? Which one will result in the smallest synthesized circuit? These are not easy questions to answer. Style 2 is both compact and allows for both Mealy and Moore machines, so this is a good all-around style to use. As for synthesized results, state encoding will have a greater effect on ultimate size than any of these variations in style.

STATE ENCODING METHODS

State encoding can have a great effect on circuit size and performance, and can also influence the amount of glitching produced by a circuit. With all that said, you should note that most synthesis tools can encode or re-encode states.

The state encoding used in the previous examples is a simple sequential numbering, as shown in Table 15-2.

Table 15-2 Sequential State Encoding

State	Code
IDLE	000
PREHEAT	001
LOAD	010
COOK	011
EMPTY	100

A common sense approach to state encoding might be to assume that the *heat* output needs to be on for the states PREHEAT, LOAD, and COOK. So the states could be encoded with one of the bits set for all of those states. This would have the effect of simplifying the output logic. The *heat* output is now simplified to *state[2].*

Table 15-3 Mapping State Code To Simplify Outputs

State	Code
IDLE	000
PREHEAT	100
LOAD	111
COOK	110
EMPTY	001

Another approach that may minimize glitching is to "Gray code" the state encoding. "Gray code" is another method of binary counting; in Gray code during each transition, only one bit changes. This is easy for some state machines and difficult or impossible for state machines that branch in many directions. For the automatic oven, we could encode the states as shown in Table 15-4.

Table 15-4 Gray State Encoding

State	Code
IDLE	000
PREHEAT	100
LOAD	110
COOK	111
EMPTY	101

In this encoding between each state, only one bit changes (either sets or clears). The only transition in this model that violates this Gray code rule is the transition from EMPTY to IDLE, during which two of the bits clear.

For each of the encodings shown in Table 15-2, Table 15-3, and Table 15-4, only three flip-flops are used to encode the states. Because the state machine has five states, the minimum number of flip-flops to use is three. If you are not concerned about using the minimum number of flip-flops, there are other encodings you can use.

If you want to get the outputs out as quickly as possible, we can re-encode the states to an encoding of *output = state*. To start the *output = state* encoding, let's look at all the states in reference to the outputs, as shown in Table 15-5.

Table 15-5 States Compared with Outputs

State	LOAD	HEAT	UNLOAD	BEEP
IDLE	0	0	0	0
PREHEAT	0	1	0	0
LOAD	1	1	0	0
COOK	0	1	0	0
EMPTY	0	0	1	1

Now we can add a state encoding to Table 15-5, yielding the data in Table 15-6.

Table 15-6 Outputs as State Code

State	State Code	LOAD	HEAT	UNLOAD	BEEP
IDLE	00000	0	0	0	0
PREHEAT	01000	0	1	0	0
LOAD	11000	1	1	0	0
COOK	01010	0	1	0	0
EMPTY	00101	0	0	1	1

This state encoding has a bit for each output and extra bits for states that do not have unique outputs. The states PREHEAT and COOK both have the same outputs, so they need to have two different encodings. The last bit, used for the BEEP output, can be removed for optimization since it is the same as UNLOAD. This method of encoding uses four or five flip-flops, so it yields a larger circuit than do the other encoding methods.

A final state encoding method is called "one-hot." In this encoding method there is exactly one flip-flop set per state. This method may take even more flip-flops, but can sometimes produce faster circuits. One-hot state encoding is shown in Table 15-7.

Table 15-7 One-Hot State Encoding

State	Code
IDLE	10000
PREHEAT	01000
LOAD	00100
COOK	00010
EMPTY	00001

Which state encoding is best for a particular design depends on your design goals. Does the design need to be fast or small? Is glitching a concern? Is simultaneous switching a concern for power and glitching? These are the questions that most influence the choice of a state encoding.

DEFAULT CONDITIONS

Each of the state machine examples included a *default* clause. Those examples used 3 bits for state, but only five states were used. Thus it may be possible for the state machine to glitch into one of the three remaining illegal states. One other reason for

including the *default* clause is that when simulation starts, the state machine is in an unknown state, and the *default* clause gets it on track with the first clock.

The *default* clause may cause synthesis to generate more logic, so a trade-off must be made between the security of having a *default* clause and the potential size savings of not having it.

IMPLICIT STATE MACHINES

In all of the previous state machine examples, the three sections of the state machine were obvious (or explicit) in the coding style. The automatic oven design could also be coded as an implicit state machine. This style can be easier to code and maintain in an abstract model. In this style, only the behavior of the state machine is seen. The values of the outputs can also be seen, but the state register and next-state logic are implied.

Example 15-8 Implicit State Machine Style

```
module auto_oven_implicit(clock, start, temp_ok, done, quiet,
  load, heat, unload, beep);
input clock, start, temp_ok, done, quiet;
output load, heat, unload, beep;
reg load, heat, unload, beep;
initial begin // set all outputs to the off state
    load = 0;
    heat = 0;
    unload = 0;
    beep = 0;
end

always begin
    @(posedge clock) ; //stay IDLE at least one clock
    while( ! start) @(posedge clock) ;
                                // do nothing until a start
    heat =1 ; // turn on heating element
    load = 1;
    @(posedge clock) ; //stay PREHEAT at least one clock
    while( ! temp_ok) @(posedge clock) ;  // wait to heat up
    load = 1;
    @(posedge clock) load = 0;
    @(posedge clock) ;
                    // stay in COOK at least one clock cycle
    while( ! done) @(posedge clock) ;
                   // wait to finish cooking
    heat = 0;
    unload = 1;
    if (! quiet) beep = 1;
    @(posedge clock) unload = 0;
```

```
    beep =0;
end
endmodule
```

You have also already seen two other examples of implicit state machines. In chapter 6, when introducing the looping constructs, the modules *shift1* and *onecount* were simple, synthesizable implicit state machines.

REGISTERED AND UNREGISTERED OUTPUTS

The modeling of the output section of the state machine may infer that the outputs are either registered, or combinatorial. Registered outputs are available sooner after the clock and are less subject to glitching. One great disadvantage of registered outputs is that the machine becomes larger due to the extra flip-flops. Because the outputs are synchronous, this implies that machines with registered outputs are only Moore machines. Changing the Verilog for the output section to be registered instead of combinatorial is a simple matter of changing the *always* block to execute only on a clock, rather than a change on any input.

Example 15-9 Combinatorial Outputs

```
 always @(state or quiet) begin
  if(state == `LOAD) load = 1; else load = 0;
  if(state == `EMPTY) unload = 1; else unload = 0;
  if(state == `EMPTY && quiet == 0) beep = 1; else beep = 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK) heat = 1; else heat =0;
end
endmodule
```

Example 15-10 Registered Outputs

```
always @(posedge clock) begin
  if(state == ` LOAD)
    load <= #(` REG_DELAY) 1;
  else
    load <= #(` REG_DELAY) 0;
  if(state == ` EMPTY)
    unload <= #(` REG_DELAY) 1;
  else
    unload <= #(` REG_DELAY) 0;
  if(state == ` EMPTY && quiet == 0)
    beep <= #(`REG_DELAY) 1;
  else
```

```
    beep <= #(`REG_DELAY) 0;
  if(state == `PREHEAT ||
     state == `LOAD ||
     state == `COOK)
    heat <= #(`REG_DELAY) 1;
  else
    heat <= #(`REG_DELAY) 0;
end
endmodule
```

Example 15-10 will produce its outputs delayed by one clock. This may work for some machines, but it may make others out of sync.

Since it is highly desirable to have registered outputs for high speed machines or to eliminate glitching, a modified Moore machine may be used to create registered outputs in sync with the state machine.

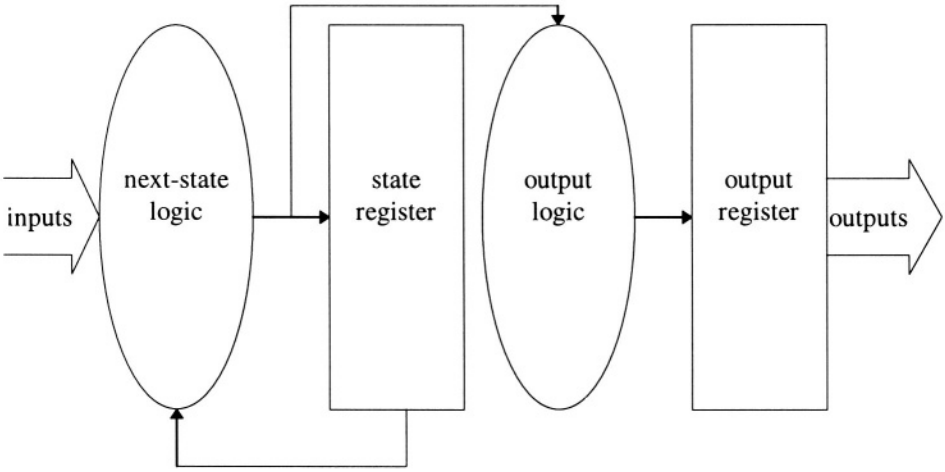

Figure 15-3 Modified Moore Machine

The modified Moore machine allows for registered outputs as shown in Example 15-11.

Example 15-11 Modified Moore Machine with Registered Outputs

```
always @(posedge clock) begin
  if(next_state == `LOAD)
    load <= #(`REG_DELAY) 1;
  else
    load <= #(`REG_DELAY) 0;
  if(next_state == `EMPTY)
    unload <= #(`REG_DELAY) 1;
  else
    unload <= #(`REG_DELAY) 0;
  if(next_state == `EMPTY && quiet == 0)
    beep <= #(`REG_DELAY) 1;
  else
    beep <= #(`REG_DELAY) 0;
  if(next_state == `PREHEAT |
     next_state == `LOAD |
     next_state == `COOK)
    heat <= #(`REG_DELAY) 1;
  else
    heat <= #(`REG_DELAY) 0;
end
endmodule
```

FACTORS IN CHOOSING A STATE MACHINE MODELING STYLE

There are many ways to model a state machine. How do you know which style to use and how to encode your states? There are a number of different goals you may have for your state machine: Maximum frequency, area, clock-to-output delay, glitch-free outputs, minimal input setup time, minimum simultaneous switching, minimal power, or ease of maintenance. Each of these goals dictates a different style or encoding method.

For a state machine to have the highest possible frequency, the next-state logic must be as small as possible. In a logic type such as CMOS (where AND gates are fast), one-hot encoding may generate the fastest next-state logic because each state bit is usually set from the outputs of only AND gates. In other state encodings, there tend to be AND/OR networks driving each state bit.

For minimal clock-to-output delay, a state machine where the outputs come directly from flip-flops is best. The output-equals-state encoding or registered output style state machines both have outputs that come directly from flip-flops, making either of these styles the best choices for state machines requiring fast clock-to-output times.

For minimal simultaneous switching and minimal power in CMOS, a state machine with the states Gray-coded might be the best solution. The Gray-coded state machines only change 1 bit per transition.

If the state machine is visualized as a flow, the implicit state machine makes it easy to model and maintain. Inserting or removing behavior from this type of state machine is easiest.

If high speed or glitch free outputs are desired, a style 1 modified Moore Machine may be the best choice.