State Machines

This lecture describes how to design state machines and implement them using Verilog.
After this lecture you should be able to: design a state machine based on an informal description of its operation, document it using state transition diagrams and tables, write a synthesizable Verilog description of it and convert between these three descriptions.

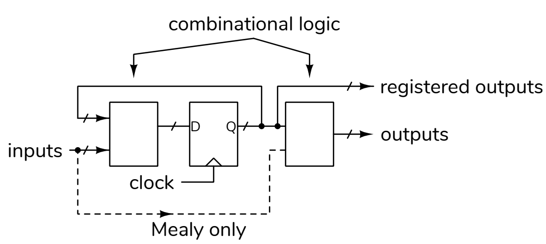
Introduction

A state machine[1] is a device whose outputs are a function of previous inputs. A state machine therefore has memory. The contents of this memory are called the "state."

Devices are often described as state machines. We will learn to describe state machines and to implement them using digital logic circuits.

Mealy vs Moore State Machines

There are two types of state machines. The outputs of a *Moore* state machine are a function only of the current state while the outputs of a *Mealy* state machine are also a function of the current inputs:



Moore state machines are simpler but changes to their outputs are delayed until the next clock edge.

Registered outputs avoid glitches resulting from different propagation delays through the combinational logic at the output. This is desirable for signals that go off-chip.

Exercise 1: Which signal in the above diagram represents the current state?

Exercise 2: Which outputs change on the rising clock edge? Which change when the input changes?

[1] An implementable state machine has a finite amount of memory and is sometimes referred to as a "finite state machine" (FSM). A state machine that implements a computational algorithm is sometimes called an Algorithmic State Machine (ASM).

Design of State Machines

The following steps can be used to design a Moore state machine. This initial design may need to be refined by adding or removing states or changing the transitions conditions until the solution meets the requirements.

Step 1 - Inputs and Outputs

The first step is to accurately identify the inputs and outputs. This is important because the rest of the design effort will be wasted if necessary inputs or outputs are not included in the design.

The outputs will typically be specified by the requirements. You should ensure the selected inputs are sufficient to provide the desired behaviour.

Step 2 - States

The second step is to identify a sufficient number of states.

Since the output of a state machine depends on the previous inputs we could – in theory – use a shift register to store previous inputs and use combinational logic to compute the current output from the contents of the shift register and the input. However, in most cases it's possible to use a much more concise representation of the states.

One approach is to begin by listing all the required *combinations* of the outputs. For a Moore state machine that has only registered outputs each of these will correspond to a state.

Exercise 3: Why?

However, the outputs of the state machine are often insufficient to define its operation. In this case we need to add "hidden" state variables which store some sort of summary of the past input values.

For example if we are interested in detecting a particular sequence of input values the state variable may be the number of items in the sequence (e.g. a password) that have matched thus far. Or if we are interested in counting the number of times an input value has appeared then the state variable may be a counter.

Exams in this course will provide hints on the choice of state variables when the choice is not obvious.

Step 3 - State Transitions

The final step is to convert the informal description or specification of the state machine's behaviour into a formal description that defines:

(i) all possible state transitions, and

(ii) the input condition(s) required for each of these transitions.

In the process of defining the transition conditions you may find that it's not possible to unambiguously determine the next output based solely on the current output and the input. This implies that there are state variables that are do not appear in the output.

This indicates the need for "hidden" states (two or more states with the same output) that allow the required state transitions to be made unambiguously. The choice of these state variables is described above.

State Machine Descriptions

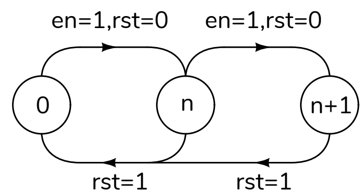
State machine are typically documented as a state-transition table or a state-transition diagram.

A state transition table is a truth table with columns for the initial state, the input condition(s), and the next state. The output corresponding to each different state (and inputs for Mealy state machine) can also be listed in the same or a different table. An example for a resetable counter with an enable input might look as follows[2]:

current state	reset	enable	next state
n	0	0	n
n	0	1	$n+1$
n	1	X	0

[2] Overflow condition omitted.

A state machine with a small number of states can be described using a state transition (or "bubble") diagram. Each circle represents a different state and arrows represent the state transitions. Each transition is labelled with the input required for that transition and each state is labelled with a state name and, for a Moore state machine, the output for that state.



Changes of state are zero-duration events that correspond to the arrows (directed edges) on a state transition diagram. In a state transition table these events are defined by a (current state, next state) pair which are the outputs of a state machine's state register before and after the event.

State transition diagrams often omit input conditions that don't result in a change of state and use `X` for "don't care" input values.

Implementation

State Encodings

In many cases, such as the counter example above, the state variables are the outputs. This has the advantage that no additional flip-flops are necessary to obtain registered outputs.

k flip-flops can be used to represent an arbitrary 2^k states. For example, 3 flip-flops could encode up to 8 states.

FPGA or CPLD designs often use "one-hot" encodings where one flip-flop is used for each state and only one flip-flop at a time may set to 1. This encoding requires more flip-flops but can simplify the combinational logic.

Exercise 4: If we used 8-bits of state information, how many states could be represented? What if we used 8 bits of state but used a "one-hot" encoding?

State Transition and Output Logic

The state transitions are implemented as combinational logic that computes the next state based on the current state and the input. In Verilog this can be done using `assign` or `always_comb` statements.

Outputs that are not represented by state variables must be computed by combinational logic from the state and, in the case of a Mealy state machine, the inputs.

A practical circuit also needs a clock signal and a reset input. The FSM will change state on every rising edge of the clock and revert to a starting state when the reset input is asserted. Often the reset is synchronous – it is an input and the circuit transitions unconditionally to the required state on the next rising edge of the clock. An example is shown above.

Multiple State Machines

Most systems contain multiple state machines interacting with each other. Each one may have different state transition rules and their state transition diagrams can be drawn separately.

For example, a multi-digit counter may be designed as a combination of individual single-digit counters each designed as a state machine with a terminal-count output and a count-enable input. A one-digit BCD counter might respond to the transition from 9 to 0 of the next-lower-order digit.

Another example would be traffic light. The transitions between light states would be controlled by a timer which is a state machine. The timer might be set or reset on a transition between traffic light states.

Exercise 5: The link below describes a game. List the top-level game states. Decompose each of these into multiple states. Repeat.

Simon Game

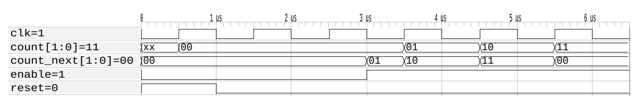
Examples

Counter

In this example the state is the counter output. The state transition table, the System Verilog model and simulation waveforms for a 2-bit counter with reset and enable inputs are shown below.

count				next count	
[1]	[0]	reset	enable	[1]	[0]
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	0	1	0	0
a	b	0	0	a	b
X	X	1	X	0	0

```
// 2-bit counter with enable and
// synchronous reset

module ex22 ( output logic [1:0] count,
      input logic enable, reset, clk ) ;

   logic [1:0] count_next ;

   // next-state logic
   always_comb begin

      if ( reset )
        count_next = 2'b00 ;
      else if ( ~enable )
        count_next = count ;
      else
        case(count)
          2'b00: count_next = 2'b01 ;
          2'b01: count_next = 2'b10 ;
          2'b10: count_next = 2'b11 ;
        default: count_next = 2'b00 ;
        endcase
   end

   // register
   always_ff@(posedge clk)
     count <= count_next ;

endmodule
```



Exercise 6: What happens if both reset and enable are asserted?

Exercise 7: Draw the state transition diagram.

Exercise 8: Rewrite the state transition table and the module using n and n+1.

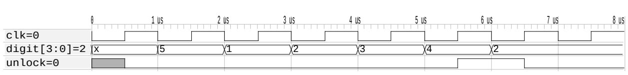
Sequence Detector

This type of state machine is used to detect a sequence of values such as the correct combination entered into a digital lock. In this case the single-bit “unlocked” output is not enough state to determine if the correct sequence has been input.

This implementation uses a shift register to store past inputs and combinational logic to detect the required pattern (1,2,3,4 in this example) in the input.

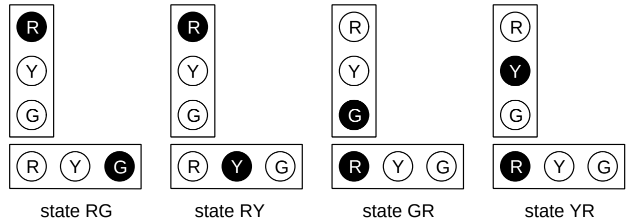
The output is registered and will be high for one clock period when the correct sequence is recognized. A practical digital lock would change state only when a key is pressed (or released) rather than on every clock edge.

```
// digit-sequence detector

module ex24 ( output logic unlock,
              input logic [3:0] digit,
              input logic clk ) ;

   logic [3:0] digits[4], digits_next[4];
   logic unlock_next ;

   // next-state logic
   always_comb begin

      for ( int i=0 ; i<3 ; i++ )
        digits_next[i] = digits[i+1] ;

      digits_next[3] = digit ;

      unlock_next = digits_next ==
           '{ 4'd1, 4'd2, 4'd3, 4'd4 } ;

   end

   // register
   always_ff@(posedge clk) begin

      digits <= digits_next ;
      unlock <= unlock_next ;

   end

endmodule
```



Traffic Lights

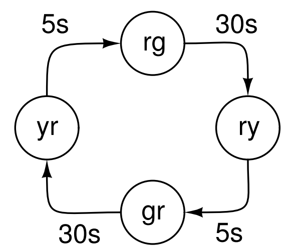
This is an example that combines two state machines: one to sequence the traffic lights at an intersection and one to implement delays. The states are encoded as the on/off values of the (Red, Green, Yellow) lights in each direction:



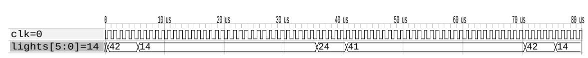
A package is used to define an enumerated type to label the four states (`rg`, `ry`, `gr`, and `gy`) according to the signal colors in the two directions:

```
package ex28pkg ;

typedef enum logic [5:0]
//           RYG RYG
     { rg=6'b100_001, ry=6'b100_010,
       gr=6'b001_100, yr=6'b010_100 }
     lightstate ;

endpackage
```

Delays are implemented by decrementing a counter on each clock edge. When the counter reaches zero the state changes and the counter is loaded with the duration of the next state.
The state transition diagram showing the duration of each state is:



The simulation outputs (with the `lights` shown in octal) are shown below:



The module definition is given below. The state and counter values are given initial values. On some technologies, these are the values when a device is powered up.

```
// traffic light controller

import ex28pkg::* ;

module ex26 ( output lightstate lights,
              input logic clk ) ;

   lightstate state=rg, state_next ;
   logic [4:0] count=0, count_next ;
```

```
   // combinational logic
   always_comb begin

      // next traffic light state
      state_next = state ;
      if ( ! count )
        case (state)
          rg: state_next = ry ;
          ry: state_next = gr ;
          gr: state_next = yr ;
          yr: state_next = rg ;
        endcase

      // duration of next state (-1)
      if ( ! count )
        if ( state == rg || state == gr )
          count_next = 4 ;
        else
          count_next = 29 ;
      else
        count_next = count-1 ;

   end

   // registers
   always_ff@(posedge clk) begin
      count <= count_next ;
      state <= state_next ;
   end

   // output
   assign lights = state ;

endmodule
```

Exercise 9: Write the state transition table for each state machine.