State Machine Design

After this lecture you should be able to:

- *design a state machine from an informal description of its operation, and*
- *write a Verilog description of a state machine.*

Introduction

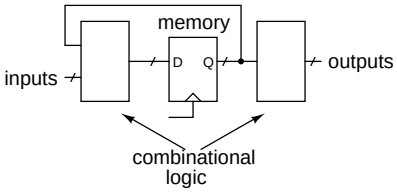
Often the clearest and most concise way to describe a system is as a state machine[1]. Thus it's important to understand how to model real-world systems as state machines and implement them using digital logic circuits.

A state machine is one whose outputs are a function of the current as well as previous inputs. Thus a state machine has memory. The current contents of this memory are the current state.

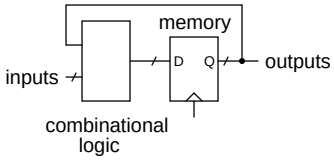
Since implementable state machines have a finite number of states (e.g. finite register widths) a state machine is sometimes referred to as "finite state machine" (FSM).

Mealy vs Moore State Machines

In digital logic design we often distinguish between two types of state machines. In a *Moore* state machine the output is only a function of the current state:

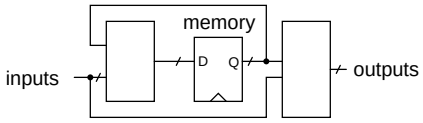


and often the outputs are themselves the state variables:



[1] This is true even when the implementation is done in software

whereas in the *Mealy* state machine the output is a function of the current state and the current inputs:



Moore state machines are simpler and are often preferred because registered outputs change only on the clock edges. These "registered" outputs prevent glitches appearing on the outputs due to different propagation delays at inputs to combination logic circuits. However, since their outputs only change on clock edges Moore state machines cannot respond as quickly to changes in the input.

Exercise 1: Which signal in the above diagrams indicates the current state?

Although in theory every flip-flop is potentially a state variable, in practice the number of states is limited – a state machine with a dozen states would be considered large – and only a few flip-flops are used as actual state variables.

In theory the output of state machine depends only on the current and previous inputs. So the canonical state machine consists of a shift register that stores all previous inputs and a combinational logic function that computes the current output from these. Although in some cases (e.g. sequence detector), this is a feasible implementation, in most cases the state variables are a more abstract and concise representation of past inputs (e.g. most recent input event, input event counts, etc).

The job of the designer is to choose specific state variables that result in an efficient (speed, area) design with, of course, correct behaviour. This requires a complete understanding of the requirements and the available implementation technology.

Hierarchical State Machines

The number of possible states increases exponentially with the number of state variables so it becomes impractical to design digital systems as one single state machine. In practice, most systems will be composed of multiple state machines that interact with each other. For example, a state machine may depend on the value of a counter but the counter itself can be designed as a separate state machine.

Exercise 2: The link below shows a game. List the top-level game states. Decompose each of these into multiple states. Repeat.

Simon Game

Documentation

State Machines are defined by the states, the state transition rules and the outputs (a function of state only or state and input).

A state machine can be documented by listing each of these in one or more tables. For example a table showing the state transitions rules would have columns for the starting state, input condition(s), and next state. The output table can have columns for the state, inputs (for Mealy state machines) and outputs.

A common way to document a state machine is to use a diagram (graph) in which each state is represented by a circle (node) and arrows (directed edges) represent the possible state transitions. Each edge is labelled with the corresponding state transition condition and each node is labelled with a state name and sometimes the output for that state.

There are many conventions for drawing these diagrams. For clarity the diagrams below often omit certain details, particularly input conditions that result in no change of state.

Design of State Machines

Students often find it difficult to come up with a set of states and the state transitions conditions when faced with a state machine design problem. The steps given below can help you come up with an initial design for a Moore state machine. An initial design often needs to be refined by adding/removing states or changing the transitions conditions until the solution meets 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.

Step 2 - States

As a first approximation, list all possible combinations of output values. For a Moore state machine each of these will correspond to a state.

Step 3 - State Transitions

Use the description of the machine's behaviour to come up with: (i) the required state transitions, and (ii) the input condition(s) required for each of these transitions.

Step 4 - Add States

In the process of defining the transition conditions you'll often find that it's not always possible to determine the next state based solely on the current state and the input. In this case you must split up the starting state into multiple states and rewrite the relevant transition conditions. These additional states will require additional flip-flops which are not outputs.

Implementation

State Encodings

There are various ways in which flip-flop values can be used to represent states.

In many cases we can use the outputs themselves as state variables. This has the advantage that no additional flip-flops are necessary to obtain registered outputs.

Another method is to use k flip-flops to represent 2^k possible states (e.g. 3 flip-flops can encode up to 8 states) and use an arbitrary mapping from states to combinations of flip-flop values. For example we could assign arbitrary numbers to the states and use their binary encodings as the values of the k flip-flops.

An alternative encoding that is often used with FPGAs is a "one-hot" encoding. In this encoding one flip-flop is used for each state and only one flip-flop at a time is set to 1. Variants of this type of encoding include "one-cold" (where only one flip-flop is 0)

and "almost one-hot" (where there is also a state, typically the reset state, where all flip-flops are 0). The advantage of these encodings that it is not necessary to decode a binary value to determine the state.

Exercise 3: 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 is usually done using nested case and/or if/else statements to select the current state and input conditions in an `always_comb` procedural block.

If the state variables don't represent the all of the outputs it's necessary to add combinational logic to compute the outputs based on the state and (in the case of a Mealy state machine) the inputs.

Of course, a practical circuit also needs clock and reset inputs. The FSM will change state on every rising edge of the clock and revert to a starting state when the reset input is asserted.

Examples

This section gives examples of some state machines that are commonly used to practice state machine design. Each one demonstrates principles that can be applied to other problems.

Alarm

This simple state machine has two states: on (alarm sounding) and off (alarm not sounding). The inputs are sensors. When a sensor input is asserted the machine switches to the 'on' state and remains there regardless of any changes on the sensor inputs.

Exercise 4: Draw the state transition diagram.

Of course, more complex designs can have alarm timeouts, keypads to arm/disarm the alarm, modes where some (e.g. indoor) sensors are disabled, etc.

This type of state machine is often used to "latch" (remember) transient inputs.

Traffic Light Controller

This state machine has three states: Red, Yellow and Green corresponding to the possible colors of the traffic light. The state transitions are controlled by timers.

Exercise 5: Draw the state transition diagram. Make a table showing the possible output values. What type of state encoding would be most appropriate?

More complex designs can have additional inputs (e.g. vehicle sensors, pedestrian push-buttons, and different cycles for different times of the day).

This state machine demonstrates one-hot encodings and the use of a second state machine (timer) to control state transitions.

Counter

A counter is a state machine that is often used to count input events. It is often used together with other state machines. When the events are clock cycles a counter becomes a timer.

Counters sometimes use a Gray coded encoding in which only one bit changes when moving from one state to the next.

Exercise 6: Write the output encodings for a 2-bit Gray-coded counter.

Although counters and timers are not typically designed as state machines, they are found in most designs.

Sequence Detector

This type of state machine is used to detect a sequence of values in an input. For example, a sequence of bits in a packet preamble or a sequence of keypresses in a digital lock.

A simple implementation involves a shift register with parallel outputs. The input is connected to the input of the shift register and a combinational logic circuits is connected to the register outputs and detects the required pattern(s).

An edge detectors is a common type of sequence detector that detects an input sequence consisting of a low followed by a high (or vice-versa).

Exercise 7: A state machine has two inputs (A and B) with overlapping pulses. Design a state machine that detects when the rising edge of A happens before the rising edge of B.

Sequence detectors are an example where the canonical state machine implementation (remember-

ing *N* previous inputs) is a good approach. This is a general way to design state machines although the design of the combinational logic can become complex when the number of patterns to be detected is large.

Exercise 8: Which of the above state machines could you use in your project?

Example: Resetable 2-bit Counter

A resetable 2-bit counter has one input (reset) and two one-bit outputs. There are four valid combinations of the outputs (00, 01, 10 and 11) and thus (at least) four states. The transition conditions are to go from one count to the next higher count if the reset input is not active, otherwise go to the zero state.

If we use the variables R as the reset, Q0 and Q1 to represent the outputs, and Q0' and Q1' as the next output, the state transition table would be as follows:

Q1	Q0	R	Q1'	Q0'
0	0	0	0	1
0	1	0	1	0
1	0	0	1	1
1	1	0	0	0
X	X	1	0	0

where 'X' is used to represent all possible values (often called a "don't care").

In this case no additional state variables are required since each state transition is uniquely defined by the current state and the input.

Exercise 9: Write the tabular description of a resetable 2-bit counter with demultiplexed outputs (only one of the four outputs is true at any time). You can assume the counter will always be reset before being used. How does this counter compare to the previous one in terms of number of flip-flops and the complexity of the combinational logic?

Exercise 10: Draw the state transition diagram.