ELEX 2117 : Digital Techniques 2
2021 Winter Term

State Machines

This lecture describes how to design and document state machines and how to 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

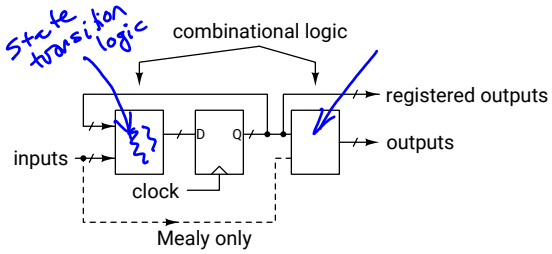
Some sequential logic circuits are most clearly described as "state machines"[1].

Although all sequential logic circuits are state machines, a state machine description is typically reserved for controllers with a small number of states. For example, although a counter is a state machine, it's typically not described as such. Examples of applications for state machines include controlling a traffic light and sequencing the calculations to compute a square root.

We will learn to describe state machines using tables, state transition diagrams and Verilog.

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 value of a state register (called the "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 clock edge following a change in the input.

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

[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).

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

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 could 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.

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.

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 obtain a much more concise description. It's often useful to start by listing the shortest sequences of inputs that result in each possible output.

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 inputs, and the corresponding next state.

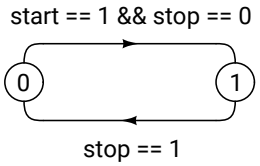
The output corresponding to each different state (and inputs for Mealy state machine) can be given in the same table or a different table.

The example below is for a motor controller with two pushbutton inputs: one to start the motor and one to stop it. A tabular description might look as follows:

inputs output

current state	start	stop	next state
0	0	0	0
1	0	0	1
X	1	0	1
X	X	1	0

A state machine with a small number of states can be described using a state transition (or "bubble") diagram. Each state is represented by a circle and arrows represent the state transitions. Each state is labelled with a state name and, for a Moore state machine, the output for that state. The transitions are labelled with the input conditions required for that transition. For the state transition table above the state transition diagram would be:



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 (e.g. from 0 to 0 or 1 to 1 above).

Implementation

State Encodings

e.g. off 00 "binary"
slow 01
fast 10

In many cases, such as the 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.

"one-hot" off 001
slow 010
fast 100

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 value of 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.

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 reset on a transition between traffic light states.

Examples

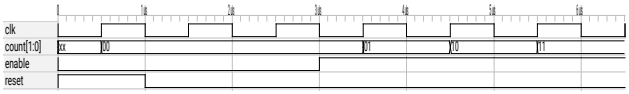
Resetable Counter

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

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

← reset
← not enable

```
// 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
   assign count_next
     = reset ? 2'b00 :
       !enable ? count :
       count == 2'b00 ? 2'b01 :
       count == 2'b01 ? 2'b10 :
       count == 2'b10 ? 2'b11 : 2'b00 ;

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

endmodule
```



Exercise 5: What happens if both reset and enable are asserted?

Exercise 6: Draw the state transition diagram.

Sequence Detector

This type of state machine detects a sequence of values such as the correct sequence of numbers for a digital lock or the sequence of sensor inputs indicating the direction in which a shaft is turning.

In the combination lock example below the single-bit "unlocked" output was insufficient to determine that the correct sequence had been input. This implementation uses a shift register to store the most recent four inputs and combinational logic to detect the required pattern (1,2,3,4 in this example) in the input sequence.

The **`unlock`** output is registered and will be high for one clock period when the correct sequence is recognized.

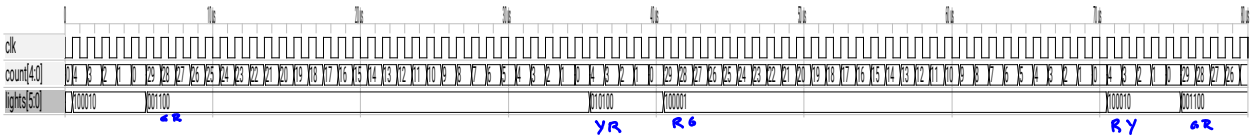


Figure 1: Simulation of traffic light controller.

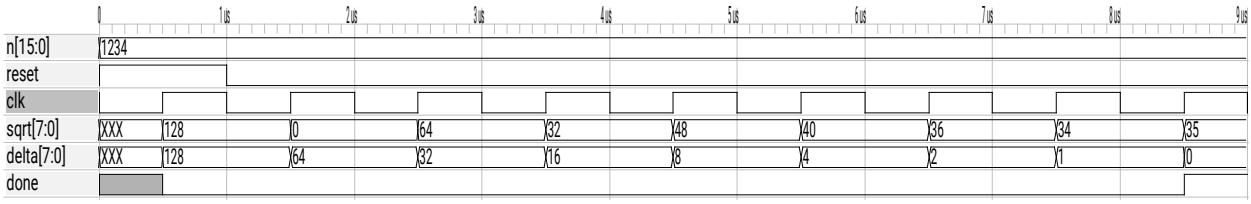
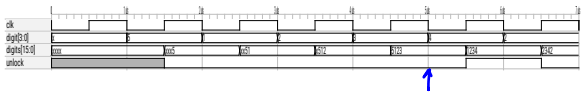


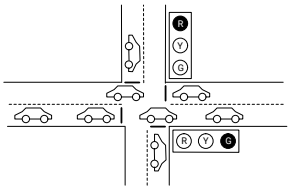
Figure 2: Simulation of the calculation of square root of 1234.

```
// digit-sequence detector

typedef enum logic { locked, unlocked } lockstate ;

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

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

   // next-state logic
   assign digits_next = digits << 4 | digit ;

   assign unlock_next
     = digits_next == { 4'd1, 4'd2, 4'd3, 4'd4 } ?
       unlocked : locked ;

   // registers
   always_ff@(posedge clk) digits = digits_next ;
   always_ff@(posedge clk) unlock = unlock_next ;

endmodule
```

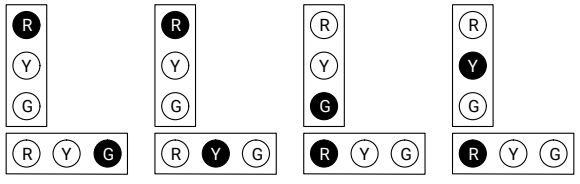


Traffic Lights

This is a controller for a traffic light at an intersection:



The controller combines two state machines: one to sequence the traffic lights and one for timing. The states are encoded as 6-bit values with the on/off values of the (Red, Green, Yellow) lights in each direction:

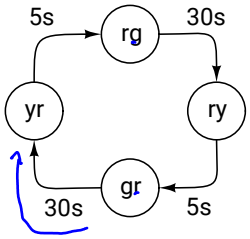


A package is used to define an enumerated type to label the four states (`rg`, `ry`, `gr`, and `gy`) according to the signal colors in the two directions:

```
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 are shown in Figure 1.

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 ;

   // next traffic light state
   assign state_next
     = count ? state :
       state == rg ? ry :
       state == ry ? gr :
       state == gr ? yr : rg ;

   // state durations
   assign count_next
     = count ? count-1 :
       state == rg || state == gr ? 4 : 29 ;

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

   // output
   assign lights = state ;

endmodule
```

Exercise 7: Write the state transition table for each state machine.

Square Root

This example computes the square root of an input number by bisection. The search interval (named **`delta`** below) could be considered to be the state variable. On reset this interval is set to half of the maximum possible value and it is divided by 2 at each iteration. The algorithm terminates when this interval is reduced to zero

```
module ex41
  (
    input logic [15:0] n,
    input logic reset, clk,
    output logic [7:0] sqrt,
    output logic done
    ) ;

   logic [7:0] sqrt_next, delta, delta_next ;

   assign sqrt_next
     = reset ? 8'd128 :
       {8'b0,sqrt} * sqrt < n ?
       sqrt + delta : sqrt - delta ;
   assign delta_next
     = reset ? 8'd128 : delta/2 ;
   assign done = !delta ;

   always @(posedge clk) sqrt = sqrt_next ;
   always @(posedge clk) delta = delta_next ;

endmodule
```

Figure 2 shows the calculation of the square root of 1234.

Exercise 8: What is the size of the expression `sqrt*sqrt`? Of `{8'b0,sqrt}*sqrt`?

Exercise 9: Draw the state transition diagram (use $\Delta = 0$ and $\Delta \neq 0$ as the states).