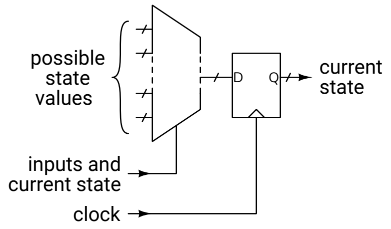
# State Machines

*This lecture defines state machines and describes how to document 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

The *state* of a logic circuit is the values of its registers. A *state machine* is a description of how the state of a circuit changes in response to inputs.

The schematic of a state machine is a register whose next value is selected by inputs, which often include the current state:



Although any logic circuit with registers could be described as a single state machine, we typically use the term when the number of possible states is relatively small – a handfull or two.

State machine descriptions are widely used. Examples of state machine applications include controllers for devices such as traffic lights or elevators; controlling the logic circuits in a CPU or GPU, and digital interfaces.

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

## State Machine Descriptions

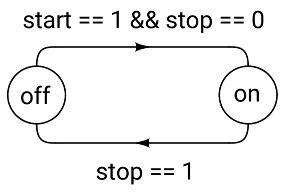
State machine are typically documented as a state-transition table or a state-transition diagram.

A state transition table is a truth table with columns for the current state, the input value, and the next state.

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:

| current state | start | stop | next state |
|---|---|---|---|
| off | 0 | 0 | off |
| on | 0 | 0 | on |
| X | 1 | 0 | on |
| X | X | 1 | off |

A state machine with a small number of states can be described using a state transition (or “bubble”) diagram. Each possible state is represented by a circle labelled with the state. Lines with arrows represent the state transitions. The transitions are labelled with the input required for that transition. For the state transition table above the state transition diagram would be:



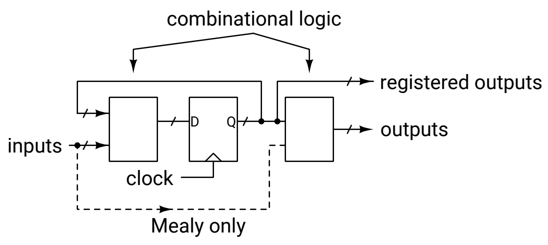
State transitions happen at the rising edge of the clock.

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).

## State Machine Outputs

A state machine is often used to generate signals that depend on the state.

For a *Moore* state machine, described above, these signals are only a function of the current state. For a *Mealy* state machine these outputs are also a function of the current inputs:



For a Moore state machine the output is a function of the state, so the number of states must be equal to or greater than the number of different outputs.

These outputs can be documented in the state transition table or diagram. For Moore state machines one output value is specified per state (one per bubble). For Mealy state machines one output value is specified per combination of state and input (one on each line between bubbles).

## Implementation

### State Encodings

$k$ 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.

For the above example the two states could be encoded as:

| state | binary encoding | one-hot encoding |
|---|---|---|
| off | 0 | 10 |
| on | 1 | 01 |

**Exercise 1:** 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?

In many cases we can choose state encodings so that bits of the state are also the outputs. Such "registered" outputs do not have glitches[1]. This is desirable for signals that go off-chip.

[1]Glitches are short-duration changes resulting from different propagation delays through the combinational logic at the output.

### 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 just another input that causes the circuit to transition unconditionally to a desired initial 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.

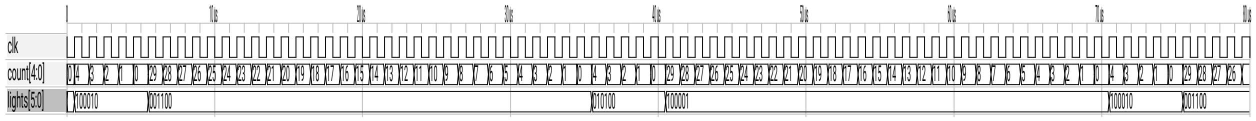


Figure 1: Simulation of traffic light controller.

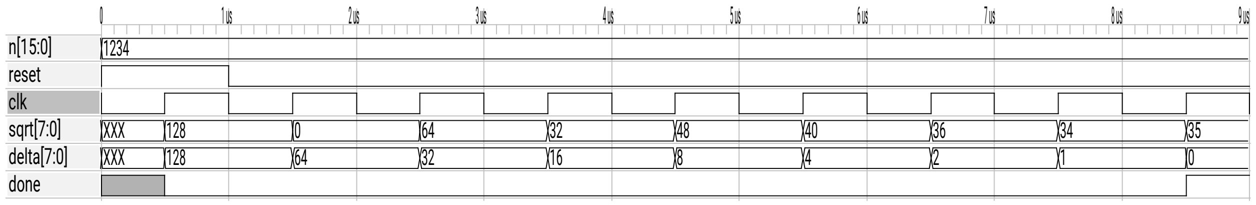
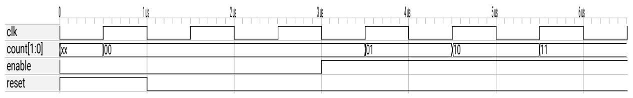


Figure 2: Simulation of the calculation of square root of 1234.

| 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 |

```
// 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 2:** What happens if both reset and enable are asserted?

**Exercise 3:** Draw the state transition diagram.

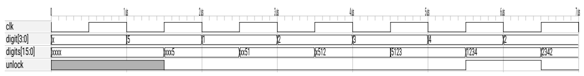
## Sequence Detector

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

The state is the most recent four inputs. Combinational logic asserts an **unlock** output when the more recent four inputs match the passcode (1,2,3,4 in this example).

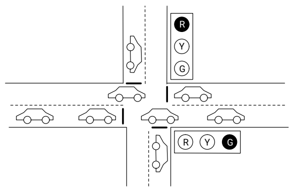
In this example the **unlock** output is registered and will be high for one clock period when the correct sequence is recognized.

```
// 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
```

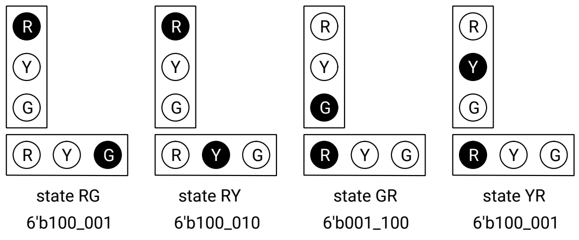


## Traffic Lights

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. A package allows us to use the same definitions in other files, for example, in a testbench.

```
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:

$$\begin{tikzcd} & rg \arrow[dr, "30s"] & \\ yr \arrow[ur, "5s"] & & ry \arrow[dl, "5s"] \\ & gr \arrow[ul, "30s"] & \end{tikzcd}$$

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 4:** 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 5:** What is the size of the expression `sqrt*sqrt`? Of `{8'b0,sqrt}*sqrt`?

**Exercise 6:** Draw the state transition diagram (use $\Delta = 0$ and $\Delta \neq 0$ as the states).