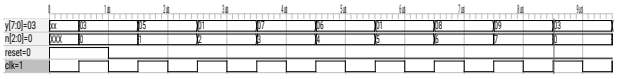
Examples of State Machines

This lecture shows how state machines can be used to design solutions for a wide variety of problems. After this lecture you should be able to write Verilog to implement: a sequence generator, a sequence detector, and circuits that combine state machines.

Sequence Generator

State machines can generate sequences. A simple example is a counter. The count value can also index an initialized array (a lookup table) to generate an arbitrary sequence:

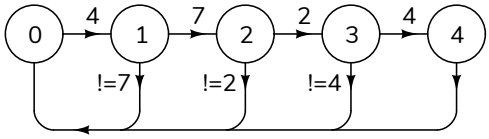
```
module ex83 ( input reset, clk,
              output logic [7:0] y ) ;

   logic [2:0] n ;
   logic [7:0] seq [0:7] = '{ 3, 5, 1, 7, 6, 1, 8, 9 } ;

   always_ff @(posedge clk) n
      <= reset ? '0 : n+1 ;

   assign y = seq[n] ;

endmodule
```



Exercise 1: How would you: (a) change the values in the sequence? (b) change the length of the sequence to 6? (c) stop at the last value?

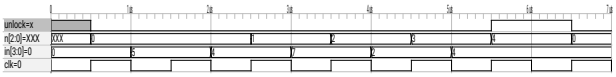
Sequence Detector

State machines can detect sequences. Consider the example of a 4-digit combination lock whose output is asserted when the most recent four inputs match the combination (4, 7, 2, 4 in this example). In this design, the state is a count of the number of correct consecutive input digits[1]:



[1]In these sequence detector diagrams the transitions are labelled with the input value(s) rather than an expression. This is common.

```
module ex82 ( output logic unlock,
              input logic [3:0] in,
              input logic clk ) ;

   logic [2:0] n ;

   always_ff @(posedge clk) n
      <= n == 0 && in == 4 ? 1 :
         n == 1 && in == 7 ? 2 :
         n == 2 && in == 2 ? 3 :
         n == 3 && in == 4 ? 4 :
         0 ;

   assign unlock = n == 4 ;

endmodule
```

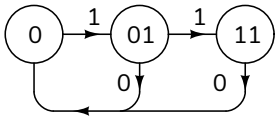


Exercise 2: Rewrite the module to store the sequence in an unpacked array as in the previous example.

Edge Detector

A sequence detector can be used to detect a rising edge on an input that is not a clock. It detects the sequence of a 0 followed by a 1.

We can implement it with three states representing: "0 or 1 followed by 0," "0 followed by 1," and "1 followed by 1":



The "rising edge detected" output would be true only in state 01.

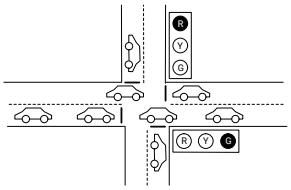
Exercise 3: Write the body of a Verilog module named **edge** with input **in** and output **out** that implements the rising-edge detector.

Combining State Machines

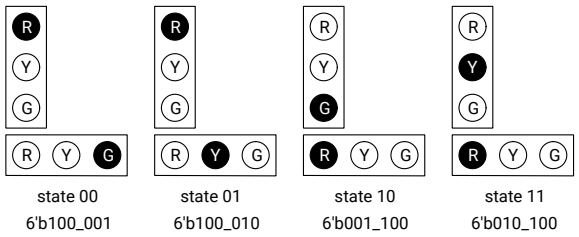
The design of a state machine can often be simplified by describing it as multiple machines. This section describes two examples of a state machine combined with a timer.

Traffic Light Controller

This example is a traffic light controller at an intersection:



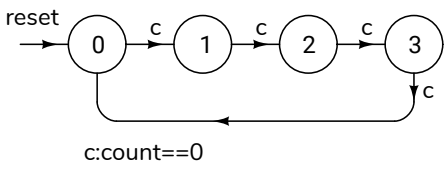
The traffic lights are controlled by a sequence generator driven by a two-bit counter named **`state`**. The state values and the corresponding **`lights`** output values are shown below:



The duration of each state is controlled by a timer. The **`count`** timer register is decremented using a 1 Hz clock.

The **`state`** register is loaded with the next state when **`count`** reaches zero and **`count`** is loaded with the duration of this next state[2].

The state transition diagram for the light state machine is:



[2]Different states have different durations.

Exercise 4: Write the state transition table for this state machine.

The timer duration depends on the state since states with yellow should be shorter. The state transition table for the timer is:

count	reset	state	next count
x	1	x	29
count $\neq 0$	0	x	count $-$ 1
0	0	00, 10	4
0	0	01, 11	29

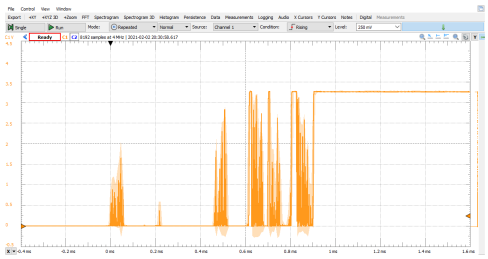
A Verilog module implementing these two state machines is:

```
// traffic light controller

module ex70
  ( output logic [5:0] lights,
    input logic reset, clk ) ;

   logic [1:0] state ; // state register
   logic [4:0] count ; // delay counter
   logic [5:0] lut [4] = '{ 6'b100_001, 6'b100_010,
                            6'b001_100, 6'b010_100 } ;

   // lights state (counter)
   always @(posedge clk) state
      <= reset ? 2'b00 :
         state == 2'b11 && count == 0 ? 2'b00 :
         count == 0 ? state + 1'b1 :
         state ;

   // set output based on state
   assign lights = lut[state] ;

   // timer
   always @(posedge clk) count
      <= reset ? 29 :
         count != 0 ? count-1 :
         state == 2'b00 || state == 2'b10 ? 4 : 29 ;

endmodule
```

The simulation results are shown in Figure 1.

Switch Debouncer

Mechanical switches “bounce” when they switch:



A switch debouncer eliminates these undesired transitions.

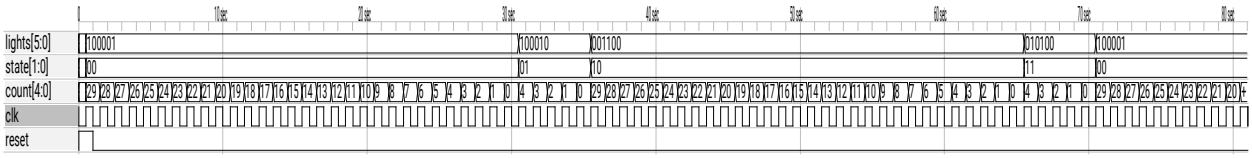
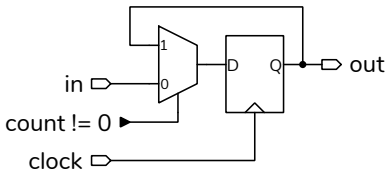


Figure 1: Simulation of traffic light controller.

The debouncer shown below also uses two state machines. The first is a one-bit state machine that holds the current output until the input is stable at a new value. The second is a timer that determines the input has been different than the output for N clock cycles.

The debouncer uses a register named **`out`** and an input named **`in`**:



The timer uses a register named **`count`**:

count	in == out	next count
x	1	$N-1$
0	x	$N-1$
n	0	$n-1$

Exercise 5: Write **`always_ff`** statements that implement these state machines.