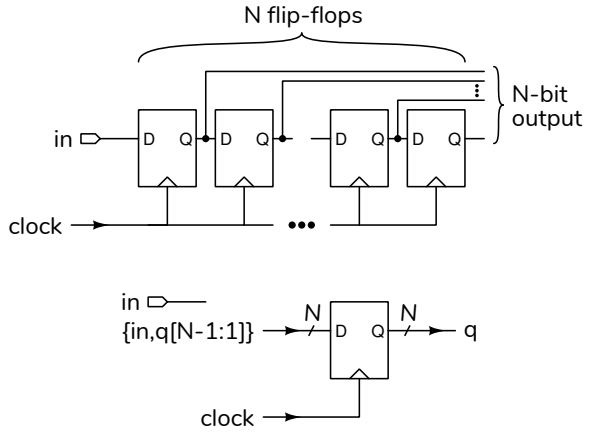
Applications of State Machines

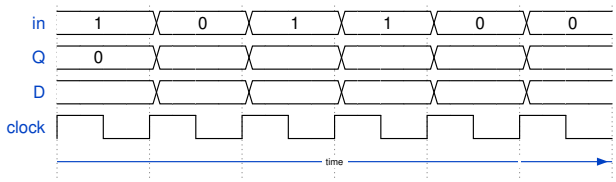
This lecture describes some applications of state machines and interacting state machines.
After this lecture you should be able to write Verilog to implement: a shift register, sequence detector, edge detector, and state machines with interdependent state transitions.

Shift Registers

A "shift register" is a register whose next value (state) is the an input bit concatenated with the current register bits. The following diagram shows two ways to draw this:



Exercise 1: The example above is an N-bit shift register that shifts the bits right. Draw a block diagram and write the Verilog for a 6-bit shift register that shifts left.



Exercise 2: Fill in the diagram above for a 4-bit ($N = 4$) right-shift shift register. Assume the initial value is zero. Which bit is the oldest (first) value in the waveform? Which bit of the shift register holds the oldest value?

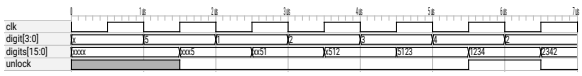
A shift register makes previous inputs available in parallel. This is useful for detecting sequences in an input.

Exercise 3: Draw a block diagram and write the Verilog for a circuit that sets an output named **`detect`** high when the sequence of values 1, 1, 0, 1 has appeared on an input named **`in`** on successive rising edges of the clock.

Sequence Detector

A sequence detector state machine can detect arbitrary sequences. In the following example, a 4-digit combination lock, the state is the most recent four input digits. Combinational logic asserts an **`unlock`** output when the most recent four inputs match the passcode (1,2,3,4 in this example).

```
// digit-sequence detector

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

   logic [0:3][3:0] digits ;

   // next-state logic

   always_ff @(posedge clk) digits
      <= { digits[1:3], digit } ;

   // sequence detector output

   assign unlock
      = digits == { 4'd1, 4'd2, 4'd3, 4'd4 } ?
        '1 : '0 ;

endmodule
```



Exercise 4: How could you modify the code so that **`digits`** is only updated when an **`enable`** input is asserted?

Exercise 5: How many states can this state machine have?

A simpler implementation would count the number of digits that had been entered in the correct order.

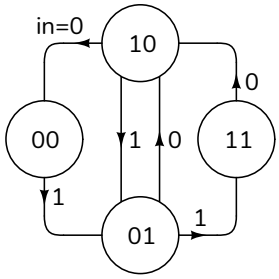
Exercise 6: Draw the state transition diagram for this simpler implementation. How many states are there? Write the Verilog using a 3-bit **`count`** state variable.

Edge Detector

An edge detector is a simple sequence detector. It detects the change of an input between clock edges. The

state consist of the value of the input at the two most recent clock edges.

The example below uses a two-bit shift register that shifts left with new bits inserted on the right. In this case the oldest bit is on the left. For example, **`10`** means the two most recent inputs were **`1`** followed by **`0`**.



Exercise 7: For which states would a **`fell`** output be asserted? A **`rose`** output? Draw the schematic and write the Verilog for this state machine. Assume an input **`in`** and a 2-bit register **`bits`** that holds the two most recent input values.

Exercise 8: Can you design an edge detector that uses only one bit? Is this a Mealy or a Moore state machine?

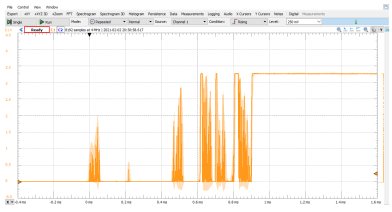
Interacting State Machines

Any circuit for which all registers share the same clock could be said to have a single register and thus could be described as a single state machine. However, it's often simpler to describe a circuit as several separate state machines. The state, or output, of one state machine can be an input to another. This section describes two examples that use a timer in addition to a simpler state machine.

Switch Debouncer

The first example is a "switch debouncer" that uses a timer to determine when to update a register.

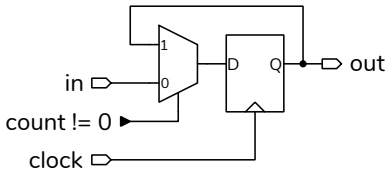
Mechanical switches "bounce" when they switch:



A switch debouncer eliminates these undesired transitions.

The debouncer shown below also uses two state machines: a timer to delay changing the output until the input has been stable for N clock cycles and a one-bit state machine to hold the current output value until the timer expires. The timer, described with a state transition table, uses a register named **`count`**. The debouncer, described with a block diagram, uses a register named **`out`** and an input named **`in`**.

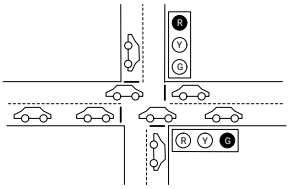
count	in == out	next count
x	1	$N-1$
0	x	$N-1$
n	0	$n-1$



Exercise 9: Write **`always_ff`** statements that implement these state machines.

Traffic Light Controller

Another example is a traffic light controller at an intersection:



One register represents which lights are turned on. Another register is the number of seconds remaining before the lights change.

One state machine sequences the traffic lights (using a register named **`state`**) and another is a timer (using a register named **`count`**). The states are encoded as 2-bit binary values. The state values and the corresponding **`lights`** output values are shown below:

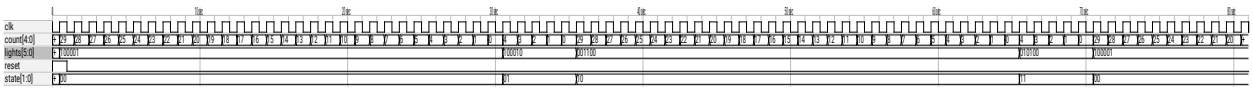
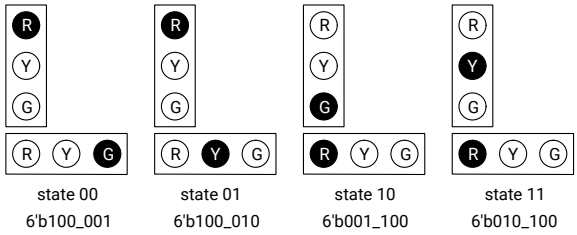
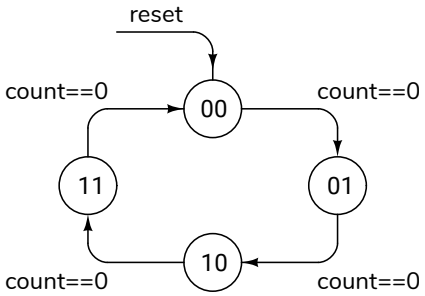


Figure 1: Simulation of traffic light controller.



Delays are implemented by decrementing **count** on each edge of a 1 Hz clock. When **count** reaches zero the **state** register is loaded with the next state and the counter register is loaded with the duration of this next state.

The state transition diagram for the light state machine is:



Exercise 10: Write the state transition table for this state machine.

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

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

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

  // set output based on state
  assign lights
    = state == 2'b00 ? 6'b100_001 :
      state == 2'b01 ? 6'b100_010 :
      state == 2'b10 ? 6'b001_100 : 6'b010_100 ;

endmodule
```

The simulation results are shown in Figure 1.