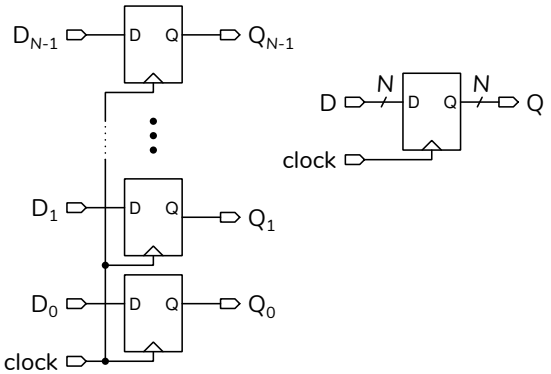
State Machines

This lecture defines state machines and describes how to document them 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

Registers

We can connect N flip-flops to the same clock to form an N-bit *register*. The common clock loads every flip-flop at the same time:

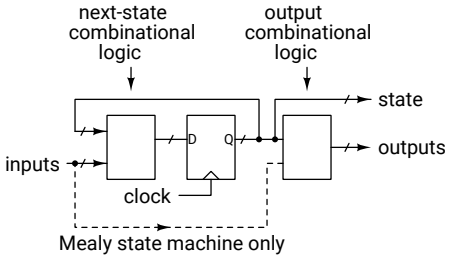


The Verilog **`always_ff`** statement creates a register.

State

The *state* of a register is its value. A *state machine* is a description of how the state changes at each rising clock edge.

The schematic of a state machine is a register whose next value is selected by a combination of the current state and, optionally, input signals:



State transitions, which are changes in the register value, only happen at the rising edge of the clock. The state register is always loaded on each rising edge of the clock, but there is no change of state if it's loaded with its current value.

The output of a state machine can be its state or a logic function such as lookup table can generate the desired output for each state.

The above describes a *Moore* state machine. A *Mealy* state machine is one where the output is a function of the current state and the inputs.

State Machine Descriptions

For example, consider a state machine with two bits of state that sequences through the values 00, 01, 10, 11 and back to 00. The output should be 0 in states 00 and 01 and 1 in states 10 and 11.

The following truth tables define this state machine's next-state and output combinational logic blocks:

state	next state
00	01
01	10
10	11
11	00

state	output
00	0
01	0
10	1
11	1

State Transition Table

A state transition table is a truth-table description of the next-state logic. It has columns for the current state, the input value(s), and the corresponding next state. Some useful conventions are: (1) x can be used for "don't care"; (2) the first matching state/input row is chosen; (3) if there is no match then there is no change of state; (4) expressions can be used to define the next state as a function of the current state and the input.

For example, if we added a reset input that set the state to 00 when it was asserted (1), we could write

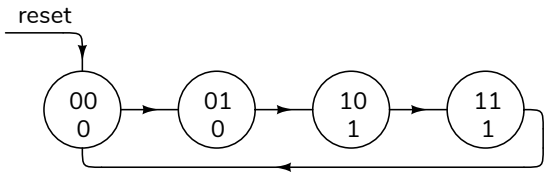
the state transition table for the counter as:

state	reset	next state
xx	1	00
11	0	00
n	0	$n+1$

State Transition Diagram

A state machine can also be described by a state transition diagram drawn using the following conventions: (1) each state is represented by a circle labelled with the state value (or name); (2) each state shows the output for that state (unless the state is the output); (3) arrows show possible transitions between states; (4) transitions happen when the expression is non-zero (true); (5) unlabelled transitions happen unconditionally; (6) transitions with no origin come from all states; (7) conditions that don't cause a state change are not shown[1]; (8) only one transition out of a state may be true (the conditions must be mutually exclusive).

The following is a state transition diagram for the state machine above:



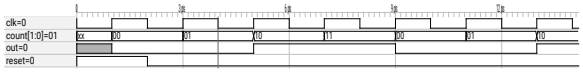
Exercise 1: Modify the diagram so the state machine counts to 11 and stops. Add a down input that cause the values to count down.

State Machines in Verilog

A state machine can be written in Verilog using one conditional operator for each transition in the diagram or table. The condition is true if the state and input value(s) match the values for that transition. The true value is the next state for that transition. The false value is the next conditional operator or, if no transition matches, the current state.

The Verilog for the example above would be:

```
// 2-bit clock divider with reset

module ex67
   ( output logic [1:0] count,
     output logic out,
     input logic reset, clk ) ;

   always_ff @(posedge clk) count
      <= reset ? 2'b00 :
         count == 2'b11 ? 2'b00 :
         count + 1'b1 ;

   assign out = count == 2'b10 || count == 2'b11 ;

endmodule
```



Counters

A counter is a common state machine. Typical inputs include those to restart the sequence (typically called reset), to pause or continue the sequence (hold or enable), or change the order of the values generated (e.g. up/down).

Exercise 2: Show the state transition diagram and table for a 2-bit counter with reset, enable, and down inputs. Reset should have priority. Write the Verilog.

Timers and Clock Dividers

It takes N clock periods for a counter to count down[2] from $N-1$ to 0. If the clock period is T then the time taken is NT. A circuit can thus create a delay of NT seconds by counting N clock cycles.

Exercise 3: What value of N would result in a 20 ms delay if the clock frequency is 50 MHz? How many bits are needed for this timer's register?

If the counter is reset to $N-1$ when it reaches 0 then the count values will be periodic with a period NT. If some event happens each time the count reaches a specific value (e.g. 0) then this event happens with period NT (a frequency $1/NT$).

Exercise 4: Assume the timer above is reset to $N-1$ each time it reaches 0. For how long is the register value 0? What are the period and frequency of a signal that is inverted each time the count reaches 0?

Interacting State Machines

Circuits often contain multiple state machines. The output of one state machine can be an input to another.

[1] Some authors do not allow this.

[2] Timers traditionally count down from $N-1$ to 0 rather than up from 0 to $N-1$ because it's simple to determine when the count reaches 0: adding -1 does not cause a carry.

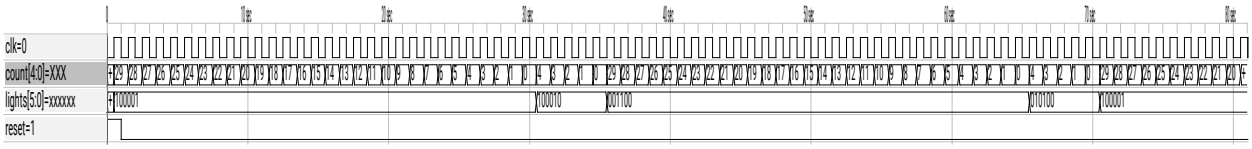
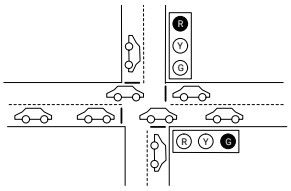


Figure 1: Simulation of traffic light controller.

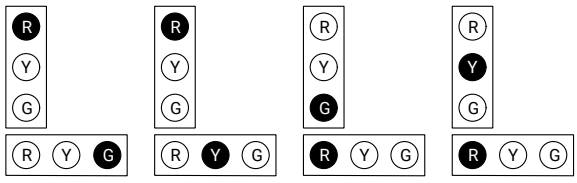
An example is the traffic light controller example below. One register represents which lights are turned on. Another register is the number of seconds remaining before the lights change.

Traffic Light Controller

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

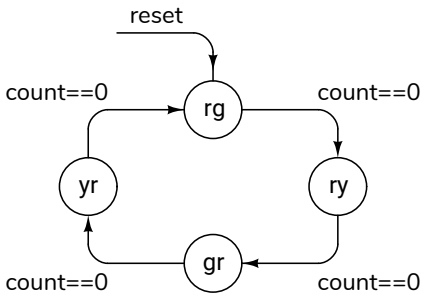


It combines two state machines: one to sequence the traffic lights (using a register named **`lights`**) and one as a timer (using a register named **`count`**). The **`lights`** states are encoded as 6-bit values with the on/off values of the (Red, Green, Yellow) lights in each direction:



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

The state transition diagram for the light state machine is:



and the state transition table for the timer is:

Exercise 5: Write the state transition table for the state machine for the **`lights`** output.

count	reset	lights	next count
x	1	x	0
$n \neq 0$	0	x	$n-1$
0	0	rg, gr	4
0	0	ry, yr	29

A Verilog module implementing these two state machines is:

```
// traffic light controller

module ex54
  ( output logic [5:0] lights,
    input logic reset, clk ) ;

   typedef enum logic [5:0] // RYG RYG
                { rg=6'b100_001, ry=6'b100_010,
                  gr=6'b001_100, yr=6'b010_100 }
                lightstate ;

   logic [4:0] count ;

   // next traffic light lights
   always @(posedge clk) lights
      <= reset ? rg :
         count ? lights :
         lights == rg ? ry :
         lights == ry ? gr :
         lights == gr ? yr : rg ;

   // state durations
   always @(posedge clk) count
      <= reset ? 29 :
         count ? count-1 :
         lights == rg || lights == gr ? 4 : 29 ;

endmodule
```

An enumerated type, **lightstate**, allows us to use more meaningful symbolic names (**rg**, **ry**, **gr**, and **gy**) for the four possible values of the **state** register.

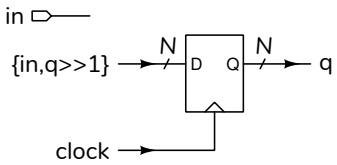
The simulation results are shown in Figure 1.

Detecting Sequences

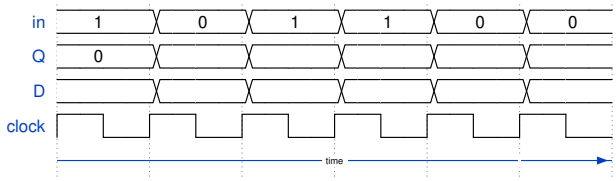
State machines are often used to detect sequences of values in an input.

Shift Registers

A shift register is an N-bit register whose input is the concatenation of an input bit and the shifted output:



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



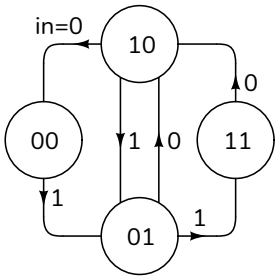
Exercise 7: Fill in the diagram above for a 4-bit ($N = 4$) right-shift shift register. Assume the initial value is zero. Which bit is the oldest (first) value in the D waveform? Which bit of the shift register holds the oldest value?

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

Exercise 8: 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.

Detecting Changes

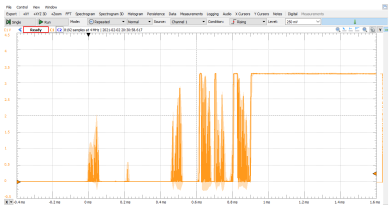
An edge detector is the simplest sequence detector. It detects the change of an input between clock edges. The state is the input at the two most recent clock edges. It can be implemented with a two-bit shift register shifting bits in from the left. For example, **10** means the two most recent inputs were **1** followed by **0**. The **rising** output is true when the input changed from low to high. The **falling** output is true for the opposite change.



Exercise 9: For which states would **falling** be asserted? **rising**? 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.

Detecting Durations

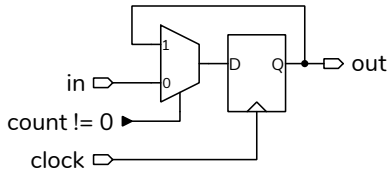
Mechanical switches "bounce" when they switch:



A switch debouncer eliminates these undesired transitions.

The debouncer shown below 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	$n-1$

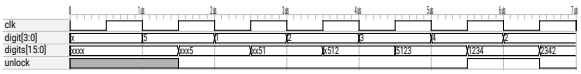


Exercise 10: Write `always_ff` statements that implement these state machines.

Detecting Sequences of Values

A sequence detector can detect longer 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 11: How could you modify the code so that `digits` is only updated when an `enable` input is asserted?

Exercise 12: 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 13: Draw the state transition diagram for this simpler implementation. How many states are there? Write the Verilog using a 3-bit `count` state variable.

One-Hot State Encodings

"One-hot" state encodings use one flip-flop per state and only one flip-flop at a time may be set to 1. This requires more flip-flops but may require less combinational logic. For the first state machine example (a 2-bit counter) the two states could be encoded in two different ways:

binary encoding	one-hot encoding
00	1000
01	0100
10	0010
11	0001

Exercise 14: How much logic is required to detect a state when a binary encoding is used? With a one-hot encoding?

Exercise 15: 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?