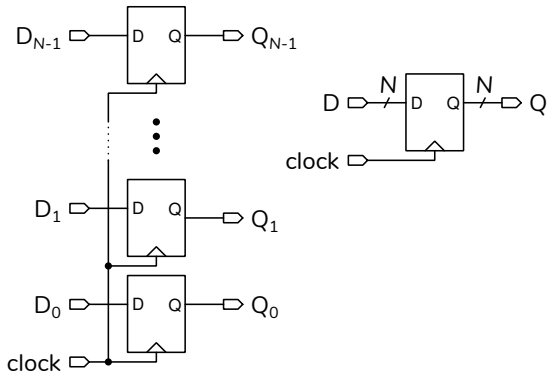
State Machines

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:

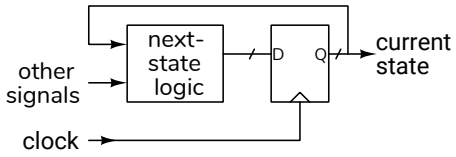


The Verilog `always_ff` statement creates a register.

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) other signals:



State Machine Descriptions

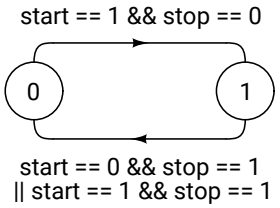
State machines can be described by state-transition tables or state-transition diagrams.

A state transition table has columns for the current state, the input value(s), and the corresponding next state.

The example below is for a motor controller with one bit of state and two inputs: one starts the motor and one stops it. The motor stops if both are asserted:

current state	inputs		next state
	start	stop	
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

A state machine can also be described using a state transition diagram. Each state is represented by a circle labelled with the state. Lines with arrows show transitions between states and the input conditions needed for that transition. The state transition diagram for this state machine would be:



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 it might be loaded with the previous value, resulting in no change of state.

The description of the state machine should be unambiguous. Two conventions that we'll use are that unlabelled transitions always happen and that transitions that don't result in a change of state (e.g. from 0 to 0 or from 1 to 1) may be omitted[1].

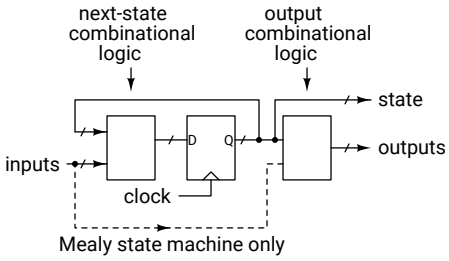
Exercise 1: Draw the schematic for this state machine. Write an `always_ff` statement that implements this state machine.

State Machine Outputs

We can often choose the register values used to represent each state (the "state encodings") to be the same as the required output for that state (e.g. motor on=1 and motor off=0).

When this is not possible we can define logic to generate the required output values for each state.

The above describes a *Moore* state machine. If the state machine output must change as a function of a signal that is not the state, then logic that generates the output will be a function of the current state and these other signals. This is called a *Mealy* state machine.



Interacting State Machines

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

For example, a multi-digit BCD counter may contain several single-digit counters. The value of a counter would increment when the next-less-significant digit value was 9.

Another example would be the traffic light controller example below. One register represents which lights are turned on. Another register is the number of seconds remaining before the transition to the next state.

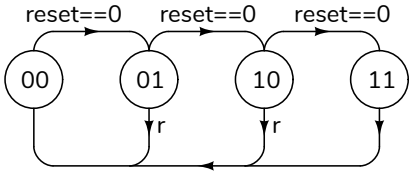
[1]Some authors do not allow this.

Sequence Generators

Sequence generator state machines produce a desired sequence of values. Typical inputs may include those to restart the sequence (typically called reset), pause or continue the sequence (hold or enable), or change the values generated (e.g. up/down, increment, or maximum value inputs).

Counter

The state transition state diagram for a two-bit counter with a reset input is:



A simple way to convert a state transition diagram to Verilog is to write one conditional expression for each transition in the diagram. This expression includes the starting state and the transition condition. Only one of these logical expressions will be true and so they can be written in any order.

If an expression is true then the state is set to the ending state for that transition. If none of the conditions match then the state is set to its current value and so it doesn't change.

```
// 2-bit counter with reset

module ex53
   ( output logic [1:0] count,
     input logic reset, clk ) ;

   always_ff @(posedge clk) count
      <= count == 2'b00 && !reset ? 2'b01 :
         count == 2'b01 && !reset ? 2'b10 :
         count == 2'b10 && !reset ? 2'b11 :
         count == 2'b11           ? 2'b00 :
         count == 2'b01 &&  reset ? 2'b00 :
         count == 2'b10 &&  reset ? 2'b00 : count ;

endmodule
```

Two conventions can simplify state transition tables: (1) An x ("don't care") can replace values that don't affect the next state. Each x halves the number of lines needed. (2) Arithmetic expressions can replace lines where the next state can be computed from the current state or the input.

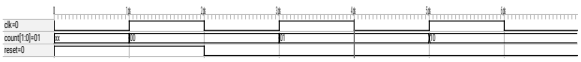
We can write the state transition table for this counter in 3 lines rather than 8 as:

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

and the Verilog would be:

```
// 2-bit counter with reset

module ex56
   ( output logic [1:0] count,
     input logic reset, clk ) ;

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

endmodule
```

Exercise 2: Simplify the solution using the fact that addition of 1 to 3 "wraps" a 2-bit value to 0. What would you change so the counter does not "wrap around" from 2'b11 to 2'b00?



State transitions diagrams are impractical if there are many states, such for a counter. A state transition table using expressions, as above, is a practical alternative.

Timers and Clock Dividers

We can create delays by counting clock cycles. 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 delay is NT.

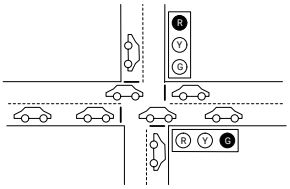
Exercise 3: What value of N would give a 20 ms delay if the clock frequency is 50 MHz? How many bits are needed for this timer's register?

If the counter is reset to $N-1$ when it reaches 0 then the count values will be periodic with a period NT.

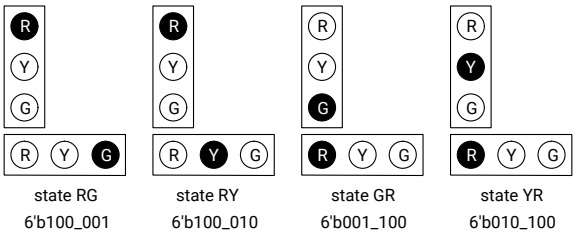
Exercise 4: Assume the timer above is reset to $N-1$ each time it reaches 0. For how long is the register value 0? What are the period and frequency of a signal that is inverted each time the count reaches 0?

Traffic Light Controller

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

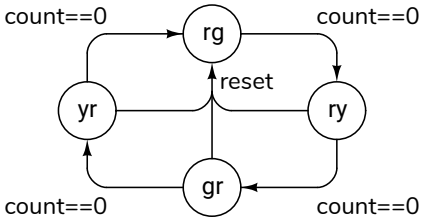


It combines two state machines: one to sequence the traffic lights and one as a timer. The states are encoded as 6-bit values with the on/off values of the (Red, Green, Yellow) lights in each direction:



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

The state transition diagram for the light state machine is:



and the state transition table for the timer is:

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 this state machine is:

```
// traffic light controller

module ex54
  ( output logic [5:0] lights,
```

[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.

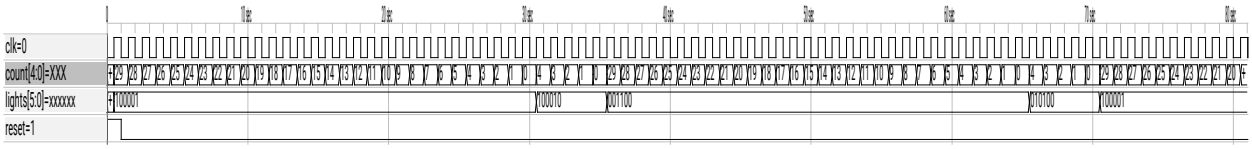


Figure 1: Simulation of traffic light controller.

```
    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.

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

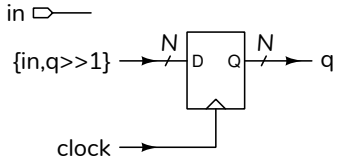
Sequence Detectors

Sequence detectors detect specific sequences of values in the input. They typically have more states than distinct output values. Examples include detecting a change in an input, a specific sequence of input values or the duration of an input.

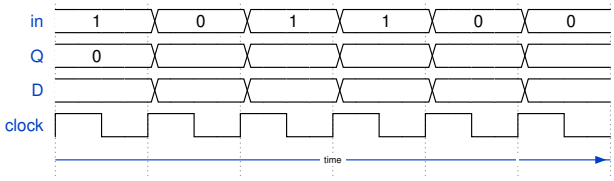
The most general implementation of a sequence detector is a shift register that stores previous inputs. Outputs are generated based on previous inputs – the values in the shift register. However, a simpler implementation is often possible as shown in the examples below.

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 Verilog for a 6-bit shift register that shifts left.



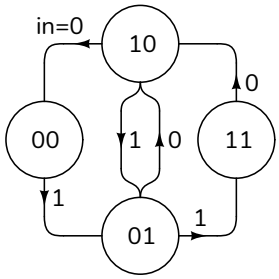
Exercise 7: Fill in the diagram above for a 4-bit ($N = 4$) right-shift shift register. Assume the initial value is zero. Which bit is the oldest (first) value in the D waveform? Which bit of the shift register holds the oldest value?

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

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. 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. store the

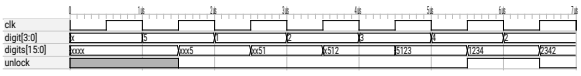


Exercise 9: For which states would **falling** be asserted? **rising**? Draw the schematic and write the Verilog for this state machine. Assume an input **in** and a 2-bit register **bits** that holds the two most recent input values.

Detecting 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 10: How could you modify the code so that **digits** is only updated when an **enable** input is asserted?

Exercise 11: 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.

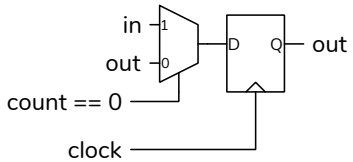
Exercise 12: Draw the state transition diagram for this simpler implementation. How many states are there? Write the Verilog using a 3-bit **count** state variable.

Detecting Durations

Most mechanical switches briefly interrupt the connection when they switch. A switch debouncer eliminates the extra edges on a switch input.

The debouncer below uses two state machines: a timer to delay changing the output until the input has been stable for N clock cycles and one to hold the current output value until the timer expires. The timer uses a register named **count** and the input and output are named **in** and **out** respectively.

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



Exercise 13: Write **always_ff** statements that implement these two state machines.

RTL Controller

"Algorithmic state machines" control the computation of numerical results. A state machine, called the the "controller," controls the values loaded into registers, called the "datapath," which hold numerical values. This is called Register Transfer Level (RTL) design.

This example computes the square root of an input number by bisection.

The datapath contains two registers: one holds the current best estimate of the square root, **sqrt**, and the other the search interval, **delta**. On reset, **sqrt** is loaded with the number whose square root it to be computed and **delta** is set to half of the maximum value. The controller state machine consists of a one-bit register, **done**, which is reset to 0 and set to 1 when **delta** reaches 0.

Note that **delta_next**, the input to the **delta** register, is a separate signal. This allows **done** to be set to **1** at the same clock edge as **delta** is set to zero.

```
module ex41
  (
```

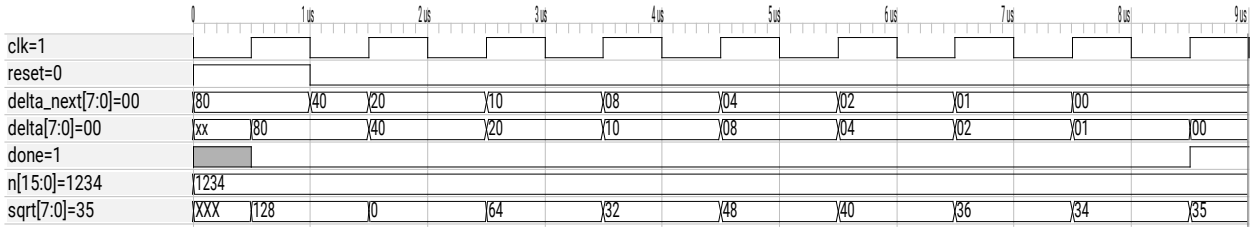


Figure 2: Simulation of the calculation of square root of 1234.

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

   logic [7:0] delta, delta_next ;

   always_ff @(posedge clk) sqrt
      <= reset ? 8'd128 :
         {8'b0,sqrt} * sqrt < n ?
         sqrt + delta : sqrt - delta ;

   assign delta_next
      = reset ? 8'd128 : delta/2 ;

   always_ff @(posedge clk) delta
      <= delta_next ;

   always_ff @(posedge clk) done
      <= delta_next == 0 ;

endmodule
```

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

Exercise 14: What is the size of the expression `sqrt*sqrt`? Of `{8'b0,sqrt}*sqrt`?

Exercise 15: Draw the state transition diagram for **done**.

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 example (the motor controller) the two states could be encoded in two different ways:

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

Exercise 16: 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?