ELEX 2117 : Digital Techniques 2
2020 Winter Term

State Machines

This lecture describes how to design state machines and implement them using VHDL.
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, and write a synthesizable VHDL description of it.

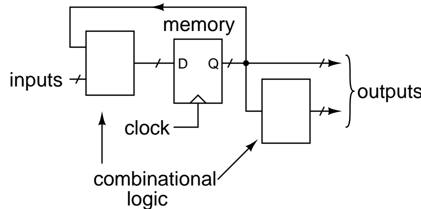
Introduction

A state machine[1] is a device whose outputs are a function of previous inputs. A state machine therefore has memory. The contents of this memory are called the "state."

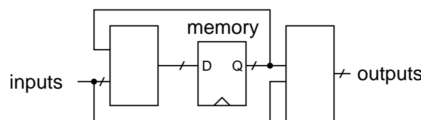
Devices are often described as state machines. We will learn to describe state machines and to implement them using digital logic circuits.

Mealy vs Moore State Machines

We can distinguish two types of state machines. The outputs of a *Moore* state machine are only a function of the current state:



whereas in the *Mealy* state machine the output is a function of the current state and the current inputs:



Moore state machines are simpler and have the advantage that "registered" outputs change only on the clock edges. This avoids glitches resulting from different propagation delays through the combinational logic at the output. This is desirable for signals that go off-chip. However, since the outputs of a Moore state machine change only on clock edges they cannot respond as quickly to changes in the input.

Exercise 1: Which signals in the above diagrams indicate the current state?

Exercise 2: Which outputs are registered? Which outputs could change whenever the input changes?

Design of State Machines

The following steps can be used to design a Moore state machine. This initial design may need to be refined by adding or removing states or changing the transitions conditions until the solution meets the requirements.

Step 1 - Inputs and Outputs

The first step is to accurately identify the inputs and outputs. This is important because the rest of the design effort will be wasted if necessary inputs or outputs are not included in the design.

The outputs will typically be specified by the requirements. You should ensure the selected inputs are sufficient to provide the desired behaviour.

Step 2 - States

The second step is to identify a sufficient number of states.

Since the output of a state machine depends on the previous inputs we could – in theory – use a shift register to store previous inputs and use combinational logic to compute the current output from the contents of the shift register and the input. However, in most cases it's possible to use a much more concise representation of the states.

One approach is to begin by listing all the required *combinations* of the outputs. For a Moore state machine that has only registered outputs each of these will correspond to a state.

Exercise 3: Why?

[1] An implementable state machine has a finite amount of memory and is sometimes referred to as a "finite state machine" (FSM).

Step 3 - State Transitions

The final step is to define the behaviour of the state machine by defining:

(i) the possible state transitions, and

(ii) the input condition(s) required for each of these transitions.

These will depend on the specifications of the state machine.

In the process of defining the transition conditions you may find that it's not possible to unambiguously determine the next output based solely on the current output and the input. This implies that there are state variables that are do not appear in the output.

In this case you must add "hidden" states (two or more states with the same output) that allow the required state transitions to be made unambiguously.

State Machine Descriptions

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

A state transitions table has columns for the initial state, the input condition(s), and the next state. The output corresponding to each different state (and inputs for Mealy state machine) can also be listed in the same or a different table.

A state machine with a small number of states can be described using a state transition (or "bubble") diagram. Each circle represents a different state and arrows represent the state transitions. Each transition is labelled with the input required for that transition and each state is labelled with a state name and, for a Moore state machine, the output for that state.

State machine descriptions often omit input conditions that don't result in a change of state and use `X` to indicate "don't care" values.

Implementation

State Encodings

In many cases we can use the outputs themselves as state variables. This has the advantage that no additional flip-flops are necessary to obtain registered outputs.

We can also use k flip-flops to represent 2^k states (e.g. 3 flip-flops can encode up to 8 states).

FPGA 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.

Exercise 4: 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?

State Transition and Output Logic

The state transitions are implemented as combinational logic that computes the next state based on the current state and the input. In VHDL this can be done using a combination of conditional and selected assignments.

If some outputs are not represented by state variables then it's necessary to add combinational logic to compute these outputs based on 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 simply another input and the circuit transitions unconditionally to the required state on the next rising edge of the clock.

Multiple State Machines

In practice, most systems will be composed of multiple state machines interacting with each other. For example, a multi-digit counter may be designed as a combination of individual single-digit counters each of which is designed as a state machine with a terminal-count output and a count-enable input.

A state machine can be designed to respond to changes of state of another state machine. For example, a one-digit BCD counter might respond to the transition from 9 to 0 of the next-lower-order digit.

These changes of state are zero-duration events that correspond to the arrows (directed edges) on a state transition diagram. These events are defined by a (next state, current state) pair which are the input and output values of a state machine's state register immediately before the event.

A state machine can also respond to the current state of another state machine. For example, an alarm signal generator might respond to the current value of an 'on' state register rather than to its transitions between on and off.

The choice depends on the desired behaviour.

Exercise 5: The link below describes a game. List the top-level game states. Decompose each of these into multiple states. Repeat.

Simon Game

Examples

Counter

In this example the state is the counter output. The state transition table, the VHDL model and simulation waveforms for a 2-bit counter with reset and enable inputs are shown below.

count				next count	
[1]	[0]	reset	enable	[1]	[0]
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	0	1	0	0
a	b	0	0	a	b
X	X	1	X	0	0

```
-- two-bit resetable counter

library ieee ;
use ieee.std_logic_1164.all ;

entity count2 is
  port (
    reset, enable, clk : in std_logic ;
    count_out : out std_logic_vector (1 downto 0)) ;
end count2 ;

architecture rtl of count2 is
  signal count, count_next : std_logic_vector(1 downto 0) ;
begin
  count_next <=
    "00" when reset = '1' else
    count when enable = '0' else
    "01" when count = "00" else
    "10" when count = "01" else
    "11" when count = "10" else
    "00" ;

  count <= count_next when rising_edge(clk) ;

  count_out <= count ;
end rtl ;
```

Exercise 6: What happens if both reset and enable are asserted?

Exercise 7: Draw the state transition diagram.

Exercise 8: Rewrite the state transition table and the code using `unsigned` signals **n** and **n+1**.

Sequence Detector

This is an example of a state machine that detects a sequence of values. In this case it is the correct combination entered into a digital lock. For a lock the output would be single bit ("unlocked"). A single bit is not enough to determine if the correct sequence has been input so this is an example where the output signals cannot be used as the state variables.

This implementation keeps track of the digit in the sequence and compares the input digit (`n`) with the expected value (3,0,1 for the three digits). If the value is correct it changes state to expect the next digit, otherwise it returns to the state for the the first digit. Once all three digits have been entered it moves to the final state where the lock is opened.

The output, **`unlock`**, will be high for one clock period when the correct sequence is recognized. A practical digital lock would change state only when a key is pressed (or released) rather than on every clock edge and would remain open until it was locked again.

```
-- combination lock

library ieee ;
use ieee.std_logic_1164.all ;

entity lock is port (
  n: in std_logic_vector (1 downto 0) ;
  clock: std_logic ;
  unlocked: out std_logic ) ;
end lock ;

architecture rtl of lock is
  signal digit, digit_next: std_logic_vector(1 downto 0);
begin

  digit_next <=
    "01" when digit = "00" and n = "11" else
    "10" when digit = "01" and n = "00" else
    "11" when digit = "10" and n = "01" else
    "00" ;

  digit <= digit_next when rising_edge(clock);

  unlocked <= '1' when digit = "11" else '0' ;
end rtl ;
```

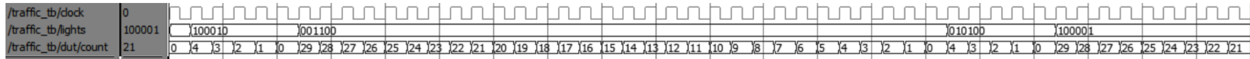
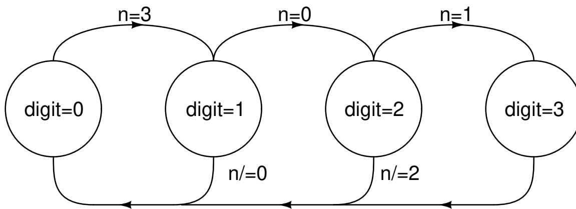
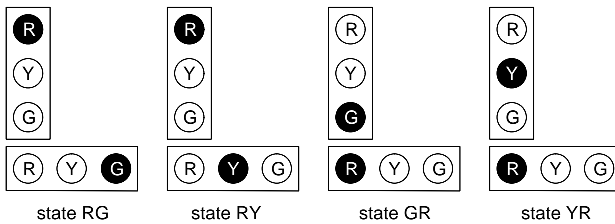


Figure 1: Simulation results for traffic light example.

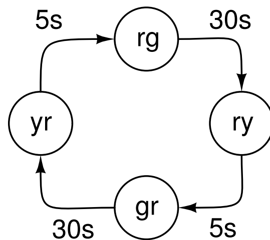


Traffic Lights

This is an example that combines two state machines: one to sequence the traffic lights at an intersection and one to implement delays. The states are encoded as the on/off values of the (Red, Green, Yellow) lights in each direction:



We use an enumerated type to label the four states (**rg**, **ry**, **gr**, and **gy**) according to the signal colors in the two directions. Delays are implemented by decrementing a counter on each (1 Hz) 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:



The VHDL description 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

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all ;

entity traffic is port (
  clock: in std_logic ;
  lights_out: out std_logic_vector(5 downto 0) ) ;
end traffic ;

architecture rtl of traffic is
  constant cbits: integer := 8 ;
  signal count, count_next, count0: unsigned(cbits-1 downto 0)
    := to_unsigned(0,cbits);
  type light_t is ( rg, ry, gr, yr ) ;
  signal lights, lights_next, lights0: light_t ;
begin

  -- light duration timer
  with lights_next select count0 <=
    to_unsigned(30-1,cbits) when rg | gr,
    to_unsigned( 5-1,cbits) when yr | ry ;

  count_next <= count0 when count = 0 else count - 1 ;

  count <= count_next when rising_edge(clock) ;

  -- light state sequencer
  with lights select lights0 <=
    rg when yr,
    ry when rg,
    gr when ry,
    yr when gr ;

  lights_next <= lights0 when count = 0 else lights ;

  lights <= lights_next when rising_edge(clock);

  -- decode state to light on/off
  with lights select lights_out <=
    "100001" when rg ,
    "100010" when ry ,
    "001100" when gr ,
    "010100" when yr ;

end rtl ;
```

The simulation outputs are shown in Figure 1.

Exercise 9: Write the state transition tables for the counter and light sequencer.