# *ENEL260 CA10: State Machines*

*Ciaran Moore*

*"Almost every object that computes is naturally viewed as a state machine"*
– Leslie Lamport

Finite state machines (FSMs) are fundamental to computing. So it's not surprising that we've seen them over and over again in the last few lectures. In this lecture we'll look at how to design finite state machines.

In CA Lecture 5, we encountered the concept of a finite state machine (FSM). In later lectures the sequence of the execution of typical microprocessor instructions was characterized in terms of FSMs, as were some peripherals. Even the humble counter can be thought of as an FSM (see Fig. 1). In this lecture we'll look in more detail at how a state machine can be designed and implemented.

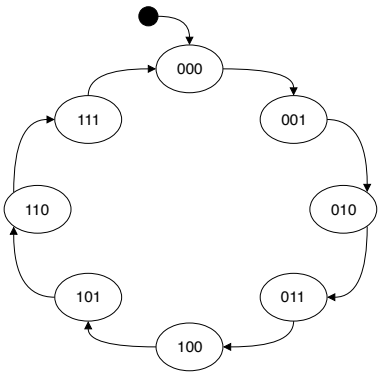



Figure 1: State diagram for a 3-bit binary counter

## *Finite State Machines*

As we saw in a previous lecture, an FSM consists of:

- A set of *states* representing whatever it is that you might want to remember about a system. It's common to specify an *initial state* as well.
- A set of *inputs*.
- A set of *transitions* (or *next-state logic*) that define how to get a *next state* based on the *current state* and the *input*.
- A set of *outputs*, and some definition of when those outputs occur (either associated with a state or with a transition).

In a *state transition diagram* (often simply 'state diagram'), like the example in Fig. 2, the states of the system are represented as bubbles, the transitions are labelled arcs, the inputs are the labels on the arcs, and the outputs are noted in the corresponding state or transition (depending on whether we're talking about a Moore or Mealy state machine).

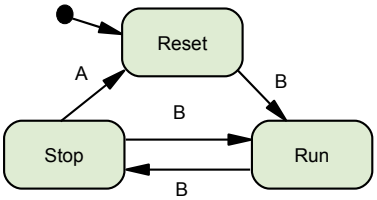



Figure 2: Example of a state machine

## *Implementing FSMs*

A Finite state machine can be implemented in hardware or in software. Software implementations of FSMs are quite common in embedded systems, graphical user interfaces, and networked communications. Hardware implementations of FSMs may be used as standalone controllers for simple embedded systems, or to offload state-driven work from a microprocessor (e.g., in a peripheral with state-based behaviour).

## *FSMs in Software*

FSMs implemented in software often[1] appear as a `switch-case` block (see the example below), or as a table lookup (see the MicroC simulator).

[1] Object-oriented designs may use some variant of the state pattern to encode the current state and next-state logic in objects.

---

*Example:* Part of an FSM implemented in C. The states and inputs are represented by constants. The `switch-case` block defines the FSM: each case implements the behaviour for a specific state, and defines the next-state logic by setting a new value for the `state` variable based on the current value of the `input` .

```
// A finite state machine
switch(state) {
  case STATE_OFF:
     if (input == POWER_BUTTON) {
        ...
        state = STATE_ON;
     }
  case STATE_ON:
     if (input == START_BUTTON) {
        ...
        state = STATE_RECORDING;
     }
     else if (input == POWER_BUTTON) {
        ...
        state = STATE_OFF;
     }
     ...
}
```

---

## *FSMs in Hardware*

When designing synchronous sequential logic FSMs, there are two common categories of state machine that you'll encounter:

- *Moore machines* treat the production of an output as depending *only* on the *current state* (see Fig. 4). When drawn as a state diagram the output is annotated along with the state, as shown in Fig. 3.

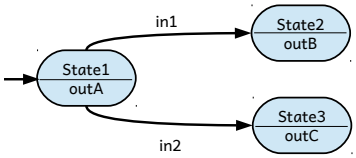



Figure 3: Outputs in a Moore machine

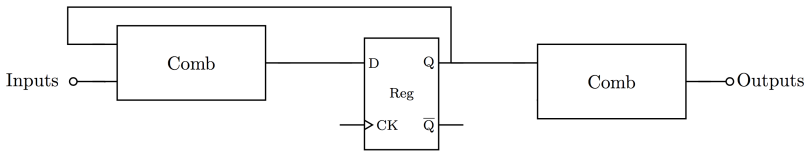



Figure 4: Moore machine (outputs depend only on the current state)

finite state machine

- *Mealy machines* treat the production of an output as depending on *both* the *current state* and the *received input* (see Fig. 6). When drawn as a state diagram the output is annotated alongside the input that triggered the generation of an output, as shown in Fig. 5.

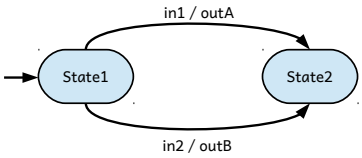



Figure 5: Outputs in a Mealy machine

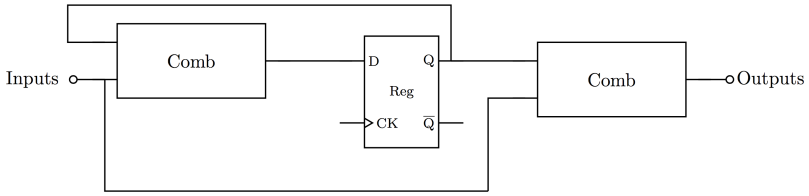



Figure 6: Mealy machine (outputs are derived from both the current state and the inputs)

Producing outputs based on both input *and* state, as in a Mealy machine, may reduce the number of states required to obtain the desired behaviour. But if the outputs are allowed to transition whenever the inputs change then the outputs may no longer be synchronized with the clock. To get around this, the inputs or outputs may need to be passed through flip-flops to ensure that they only change at predictable intervals (the tradeoff being that the additional flip-flops introduce a one clock-cycle delay).

Moore and Mealy machines have their origins in two papers from the 1950s:

- G.H. Mealy, "A method for synthesizing sequential circuits", *Bell System Technical Journal*, no. 34, 1955.
- E.F. Moore, "Gedanken-experiments on sequential machines", *Annals of Mathematical Studies*, no. 34, 1956.

## *Designing a Sequential Logic FSM*

FINITE STATE MACHINES ARE ONE OF THE FUNDAMENTAL BUILDING BLOCKS of sequential logic systems, so it pays to know how to design them.

The basic steps involved in designing a synchronous sequential logic finite state machine are:

- determining the inputs and outputs,
- capturing the required FSM behaviour,
- deciding on a state encoding,
- choosing the implementation structure, and
- designing the next-state and output logic.

1. **Determine the inputs and outputs**

   The inputs and outputs may already be specified by a requirements document or other specification. If not, you'll need to figure out what they are.

   Depending on the FSM, the inputs might represent key-presses, switch-position changes, or messages received via a network or communications port. The outputs could control LEDs and

displays, trigger network messages, or involve some other way of interacting with the outside world.

For the purposes of constructing an FSM, the inputs and outputs, whatever their source or destination, must be signals carrying logical values (i.e., they must be either *true* or *false* at any given instant). It can be helpful to list all of the inputs and outputs, and their associated meanings, in a kind of input/output "dictionary".

2. **Draw the FSM diagram**

   The state diagram should capture the required behaviour of the FSM. This may already be specified by a requirements document or other specification. If not, you'll need to figure out the required behaviour:

   - You may have specific states in mind already, and just need to connect them using inputs.
   - You may have particular relationships between inputs and outputs in mind, and will create states to implement those relationships.
   - You can play out scenarios in your head to think through what you want the state machine to do.

   Once you think you've completed the diagram, check it over to make sure it's right[2]. Ensure that you've captured the right behaviour by running one or two possible scenarios through the state diagram to see if it makes sense. Review the diagram for mistakes like ambiguous transitions (multiple transitions could be true at once) or states that can't be exited.

[2] Consider getting someone else to review it–part of the value of state diagrams is that they're a good way to *communicate* a design.

3. **Decide on a state encoding**

   Choose a binary sequence for encoding the states that has *at least* enough bits to represent all of the possible states. Create a code that assigns a unique binary sequence to each state (see lecture 2, on sequences and codes).

   Note that you don't *have* to use the minimum number of bits necessary to encode the states, although that is a common practice. Using more than the minimum number of bits can make sense in certain circumstances. For example, so-called *one-hot* encodings use one bit per state (e.g. States A, B, C would be encoded as 001, 010, and 100), which can simplify the output logic.

4. **Choose the implementation structure**

   Choose a Moore (Fig. 3) or Mealy (Fig. 5) structure, and select a number of flip-flops sufficient to store the encoded states. The structure should include (as-yet unfilled) blocks of combinational logic representing the *next-state logic* (combining inputs and the current state to generate a next state) and the *output logic* (combining the current state–and inputs if a Mealy machine–to generate an output).

The choice of Moore vs. Mealy structures is, in some ways, determined by the way the FSM diagram was drawn. However, Moore and Mealy diagrams can be converted into each other, so the form of the diagram doesn't necessarily *force* a particular choice. At the hardware level, a Moore design will tend to be more straightforward and avoids timing issues. A Mealy design may require fewer states (thus fewer flip-flops), but requires being more careful with timing. Ultimately, the choice will depend on the requirements of the system you're designing (what behaviour is needed, how important is minimizing the number of states, etc.)

5. **Design the next-state and output logic**

   Use the state diagram to construct a truth table that maps the current state and inputs to the next state[3]. Structuring the truth table to group all of the rows associate with a given state together can make it easier to relate the state diagram to the truth table. This truth table defines the *next-state* logic.

   If the outputs are not simply the current value of the state, then build a truth table that maps the state (and inputs if a Mealy machine) to the output values.

   Use the truth table(s) for the next-state and output logic to design combinational logic circuits:

   (a) Convert the truth table to a *Boolean function*, either by reading off the function directly or by using "sum-of-minterms" expression and then simplifying.

   (b) Create a *logic gate circuit* that implements the Boolean function by placing a gate for each Boolean operator and connecting up inputs to outputs through the gates.

See lecture CA4 for a refresher on designing combinational logic circuits.

[3] This is a good time to check your state diagram for completeness–does your truth table define a next-state for every possible state and input combination?
You could also just include the outputs in the state truth table, and design a single piece of combinational logic to manage next-state and output generation.

### *Extended Design Example*

Now, let's look at an example of designing a simple FSM. The task for the FSM we'll design is a controller for a *garage door opener*. The specification we've been given for the controller is as follows:

- The controller is *synchronous*, with clock pulses occurring once a second.
- There are two outputs: $Q_{mo}$ for 'motor on', $Q_{op}$ for 'opening'; thus the direction of the motor is controlled by $Q_{op}$.
- There is a 'push' input, representing the operation of the user-operated pushbutton.
- There is an 'end' input, representing the *OR* of the signals from sensors which detect when the door is *fully closed OR fully open*.
- If push occurs while the door is in the act of closing or opening, the motor is turned off; the next push causes the motor to go in the opposite direction.

Following the steps outlined above, we can arrive at a design for the controller.

1. **Determine the inputs and outputs**

   The inputs and outputs are mentioned in the specification, although only loosely defined. Based on the specification, and making some assumptions[4] about what the signals mean, we can put together the following "dictionary" of inputs and outputs:

| **Input** | **True (1)** | **False (0)** |
|---|---|---|
| push | button pushed | button inactive |
| end | door closed OR open | door between endstops |
| **Output** | **True (1)** | **False (0)** |
| $Q_{mo}$ | door moving (motor on) | door stopped (motor off) |
| $Q_{op}$ | door opening | door closing |

[4] In practice, you would confirm these assumptions with whoever provided the specification before proceeding too much further with the design.

2. **Draw the FSM diagram**

   The specification is a little hazy on the behaviour required of the controller. It tells us something about what should happen when the button is pushed, but not what should happen in response to 'end' inputs. By making some educated guesses[5] about what the controller should do (e.g., we assume that the door should stop moving when it's either fully open or full closed) we can arrive at a state transition diagram (Fig. 7) that more clearly defines the behaviour of the controller.

[5] Again, you would confirm these assumptions with whoever provided the specification before proceeding too much further with the design.

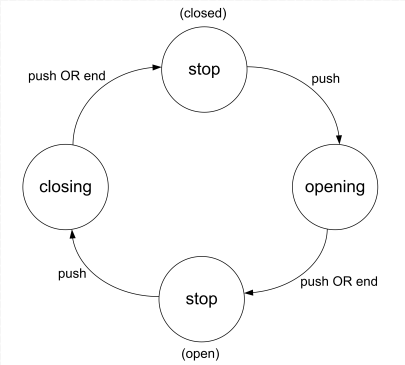



Figure 7: State diagram for the garage door opener.

   Because there are so few states, and no obvious need for asynchronous inputs, the diagram depicts a Moore FSM. All transitions from state-to-state in Fig. 7 occur *on the clock* if the stated condition is met. If the condition isn't met, then the FSM stays in its current state (i.e., there is an implicit self-loop transition).

3. **Decide on a state encoding**

   There are 4 states in the diagram, so that means we need at least a 2-bit state identifier (since $2^2 = 4$). Since the 4 states seem to correspond directly to the 4 possible combinations of outputs ($Q_{mo}\,Q_{op} = 00, 01, 10, 11$), it seems reasonable to assign the state codes so that the output is generated directly from the state (i.e., the state is $Q_{mo}\,Q_{op}$). Fig. 8 shows the state diagram with state codes assigned.

Other state code assignments are possible. Using a different encoding of the states would make the output logic more complex, but could lead to simpler next-state logic. In practice it could be worth exploring several different state encodings.

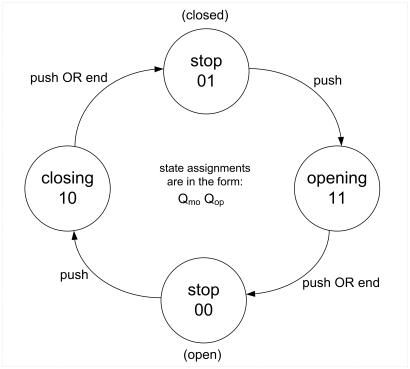



Figure 8: The state transition diagram with state codes assigned.

4. **Choose the implementation structure**

Since Fig. 8 is a Moore-style FSM, we'll use a Moore FSM structure, as depicted in Fig. 9.

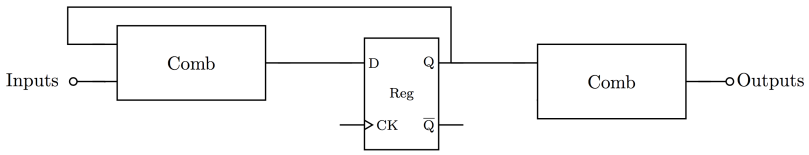



Figure 9: Moore machine (outputs depend only on the current state).

We're using 2-bit state codes, so that means we need 2 flip-flops to store the state. Since we chose the state codes[6] to align with the required outputs we don't need any *output logic*. Instead, we can use the state of one flip-flop directly as $Q_{mo}$, the output 'motor on', and the state of the other flip-flop as $Q_{op}$, the output 'opening'. The resulting design, using clocked D-type flip flops, is shown in Fig. 10.

[6] Obviously there's a little bit of an interaction between this "step" in the design process and the previous one. In practice, design is not the purely linear process presented here.

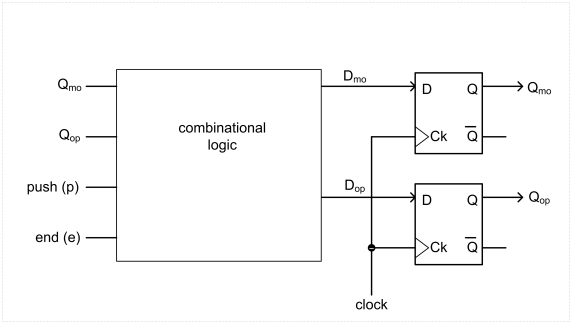



Figure 10: Moore FSM implementation for the garage door opener.

5. **Design the next-state and output logic**

Due to our choice of state encoding, the output logic is essentially already designed for us. To design the *next-state* logic to make the Moore machine function in accordance with the state diagram, we construct a truth table like the one shown in Table 1. Note that the rows of the truth table are ordered to make it easier to relate to the state transition diagram.

| Inputs | | Current State | | Next State | |
|---|---|---|---|---|---|
| $push(p)$ | $end(e)$ | $Q_{mo}$ | $Q_{op}$ | $D_{mo}$ | $D_{op}$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Table 1: Truth table for the next-state logic.

From the truth table we can form an expression for the $D$ inputs, which are the same as the next state outputs. By inspection of Table 1:

$$\begin{aligned} D_{mo} &= \overline{p}\,\overline{e}\,Q_{mo} + p\,\overline{Q}_{mo} \\ D_{op} &= \overline{Q}_{mo}\,Q_{op} + \overline{p}\,\overline{e}\,Q_{mo}\,Q_{op} + p\,Q_{mo}\,\overline{Q}_{op} + e\,Q_{mo}\,\overline{Q}_{op} \end{aligned}$$

These expressions can be converted directly into logic gate circuits that implement the controller design.

We now have a viable design for a garage door opener.

## *FSMs in the MCU*

A MICROCONTROLLER UNIT (MCU) IS CONTROLLED by a finite state machine. Now that we know something about designing FSMs, we can think about how the FSM inside an MCU might be designed.

In CA Lecture 6 we looked at a 5-step execution cycle within the microcontroller:

1. Fetch the instruction from memory at location given by the program counter PC;
   increment the program counter, PC ← PC + 1
2. Decode the instruction
3. Get Operand(s)
4. Execute operation

5. Store result

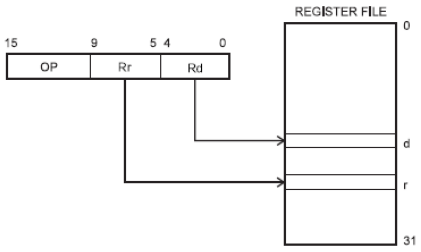
The primary *input* to the FSM is the *op code*, which will have influence over the sequence of states in order to execute different instructions. Depending on the instruction, the FSM may also look at other inputs such as the status register.

The *outputs* of the FSM will be addresses of memory locations or registers, and the control lines that change to execute an instruction, such as signals to: load the instruction register, increment the program counter, read from or write to a memory or the register file, tell the ALU which operation to execute, etc. The FSM is designed to implement the 5-step execution cycle by sequencing the signals on these control lines.

For each control line, the design of the logic subsystems they control will dictate whether they are *active HIGH* or *active LOW*. In the latter case the signal name is likely to be written with a bar over the top.

---

*Example:* Let's consider the execution of the "add with carry"' machine instruction (`ADC`) from the AVR instruction set (similar to other instructions studied in CA Lectures 6 and 7), as depicted in Fig. 11.




Operands are contained in register r (Rr) and d (Rd). The result is stored in register d (Rd).

**ADC – Add with Carry**
Adds two registers and the contents of the C Flag and places the result in the destination register Rd.

ADC Rd,Rr PC ← PC + 1

Figure 11: Example machine instruction which uses direct addressing of 2 registers

Executing the `ADC` instruction might involve control lines like[7] the following:

- $\text{WR}_{IR}$ - write to instruction register
- $\text{INC}_{PC}$ - increment program counter
- $\text{RD}_{Rr}$ - read from source register
- $\text{RD}_{Rd}$ - read from destination register
- $\text{WR}_{SR}$ - write to status register (flag bits: H,S,V,N,Z,C)
- $\text{WR}_{Rd}$ - write to destination register (from data bus)
- $\text{EN}_{AL}$ - enable ALU output to the data bus

[7] These don't necessarily resemble the actual control lines inside the AVR, but are sufficient for this example.

We can form a *timing diagram* to show the required timing of the FSM outputs, relative to each other, in order to execute the `ADC` instruction. From the timing diagram we can determine how many distinct timing points there are in the sequence, and therefore how many states the control FSM must have. One *possible* timing diagram (which assumes that all of the signals are *active HIGH*) is shown in Fig. 12. Of course, it's possible additional states would be needed for more complex instructions.

Timing diagrams are a common tool in digital logic design for documenting the timing relationships between a set of logic signals.

---

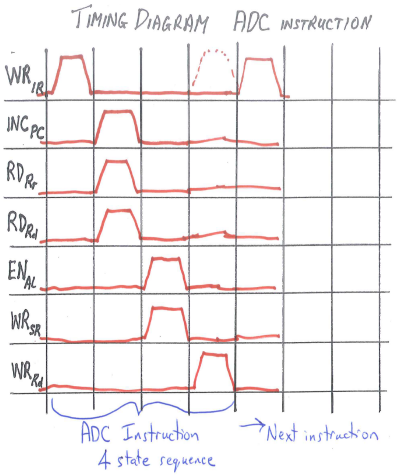



Figure 12: Timing diagram for the execution of the ADC instruction.

## *Summary*

- State machines are fundamental to computing.
- State machines form the heart of the computer itself, as well as being a common way to organize both software and internal or external hardware.
- Designing a state machine involves identifying inputs and outputs, specifying the required behaviour in terms of state transitions, and implementing the resulting state diagram by transforming it into software (switch-case statements) or hardware (next-state logic designed using truth tables).

You can learn more about state machine models of computation in Leslie Lamport's brief paper "Computation and State Machines".

---

*Looking Deeper: Converting Moore into Mealy, or Mealy in Moore*
A Moore-style state diagram can be converted into an equivalent Mealy-style diagram by taking the output associated with each state and instead placing it on *every* inbound transition to that state.

A Mealy-style diagram can be converted into an almost-equivalent[8] Moore-style diagram as follows:

- For each state that has *the same* output on *all* inbound transitions, make the output for that state equal to the output on those transitions.
- For each state that has *different outputs* on different inbound transitions, create a copy of the state for each different output. The transitions associated with a particular output should be directed to the corresponding state. The copies of the state keep the same outbound transitions as the original.

---

[8] The output appears when the transition completes, rather than on the transition.