Finite State Machines

éidhne kennedy

february 2022

What is a finite state machine?

A finite state machine, or finite state automaton, is, in broad strokes, an abstract model of computation, that can be in exactly one of a finite number of states at a given time. The machine can change from one state to another in response to some inputs; this change of state is known as a transition. [1] An FSM is defined by its list of states, its transitions between states, and its associated inputs and outputs.

So what, more precisely, beyond that surface level definition, what is a finite state automaton?

Firstly, an automaton is a mechanism designed to follow a sequence of predetermined operations, or respond to a set of predefined instructions. a finite state machine is an abstract automaton consisting of a set of "states" that the machine can occupy, a set of acceptable inputs, and a set of possible outputs. along with these three sets, the machine also contains two functions - a state transition function, which controls how the machine changes from state to state, and an output function, which, based on the result of the state transition function, computes the output of the machine. [2]

Finite state machines do not maintain memory beyond the transition between the states, and so are ideal computation models for a small amount of memory. They are not, however, capable of complex calculations like that of a Turing machine, or a modern computer. This is because although every bit of a computer can only be in two states, 0 or 1, and 2 is a finite number [3][4], however there exist an infinite number of operations that can be carried out, due to interactions between the different sets of bits.

A history of finite state machines

The first paper to present the idea of finite automata was "A Logical Calculus of Ideas Immanent in Nervous Activity"[5], which aimed to model the human thought process, aiming to simulate the firing of neurons as they perform a task in a living brain. This work was foundational to neural network theory, as well as computational theory. These ideas were expanded upon by g.h. mealy [6] and e.f. moore [7], who, in separate papers, generalised these ideas to more powerful machines. The two "styles" of finite state machine are named after them, in recognition of their work.

Finite state machines are significant today in many different fields, such as electrical engineering, philosophy, video game programming, and logic. They are also used in modeling of application behavior, the design of compilers, and in the study of formal grammar. [8]

some example finite state machines

Consider a vending machine - the input is the item selected by the user and the money put into the machine, the states defined by the amount of money currently in the machine, and the output simply the item being delivered to the user.

Or take, for example, a video game character, whose states are the action that the character is taking, and the output the corresponding animation file. The input, in this case, would be the user pressing buttons to control their character.

Or possibly a set of traffic lights at a junction, where sensors at each lane count the amount of cars waiting, and use that as the input to change the states, i.e. the lights currently displayed, and the output is the l.e.d.s in the lights actually being switched on.

Mealy and Moore machines

There are two main methods of designing finite state machines, Mealy and Moore. Both methods can be used to create any finite state machine, however both have advantages and disadvantages in different contexts. The main difference between the two styles is that the Mealy machines use both the input and the current state to calculate the output [6], and Moore machines use only the current state [7]. Note that in both machines, both the input

and current state are used to calculate the next state. This is best seen in an example, so let's take the idea of a lift that moves between the floors of a 4-story building.

Example FSM - lift

So, our lift can move between 4 floors - Ground, 1st, 2nd, and 3rd. These will be our states. Next, we need to define the inputs. In this case, it's fairly simple: an input to correspond to each state, that is to say, an input for each individual floor, that will send the lift to that floor. And finally, we need an output - let's say whether or not the doors are open.

The next step is encoding each of these inputs, state, and outputs in binary. To keep it simple, let's give the input and corresponding states the same encoding, and the output will only need to be one bit. We'll refer to the input bits as F (F_1F_0, and so on), the current state as C, the next as K, and the output as D.

For the logic, we'll operate on a simple principle: the lift will move to the floor corresponding to the input. If the input and the current floor are the same, the doors will open.

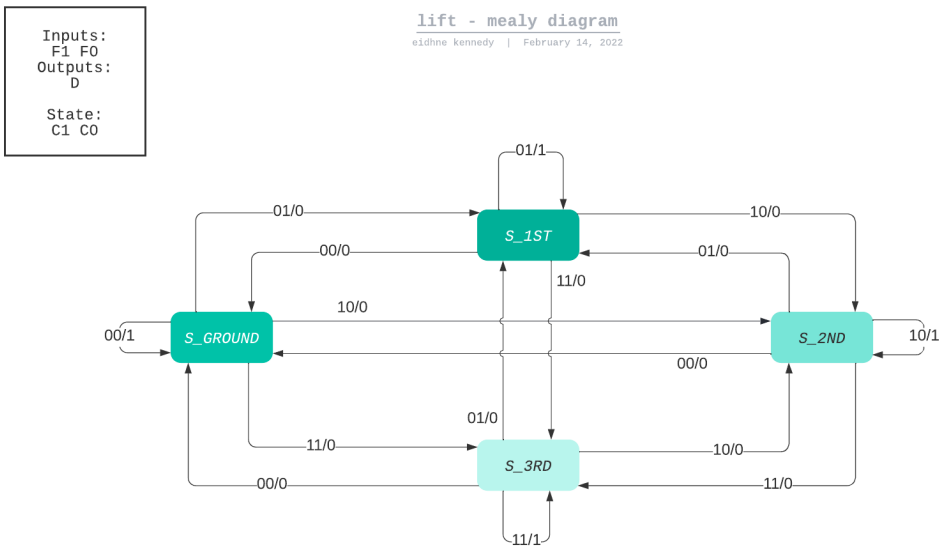
So how would we implement these as Mealy or Moore machines?

Lift - Mealy design

state	c_1c_0
ground	00
1st	01
2nd	10
3rd	11

input	f_1f_0
ground	00
1st	01
2nd	10
3rd	11

output	d
doors closed	0
doors open	1

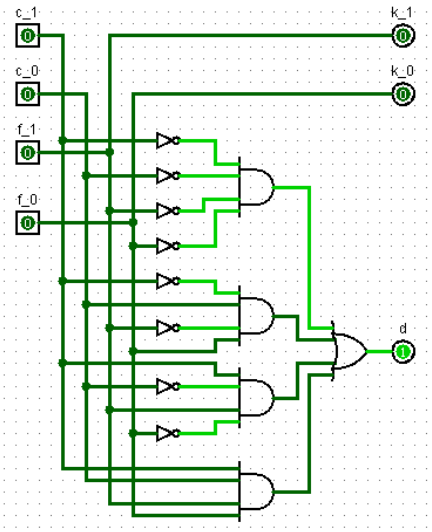


(images created using lucidchart.com[9])

We can see in the Mealy machine that we only need 4 states, as we can change the output in the self-loops that occur whenever the input and state are the same. This is reflected in the fairly simple state table, and next state logic that we implement.

c_1c_0	f_1f_1	k_1k_0	d
00	00	00	1
00	01	01	0
00	10	10	0
00	11	11	0
01	00	00	0
01	01	01	1
01	10	10	0
01	11	11	0

c_1c_0	f_1f_1	k_1k_0	d
10	00	00	0
10	01	01	0
10	10	10	1
10	11	11	0
11	00	00	0
11	01	01	0
11	10	10	0
11	11	11	1



Mealy next state logic

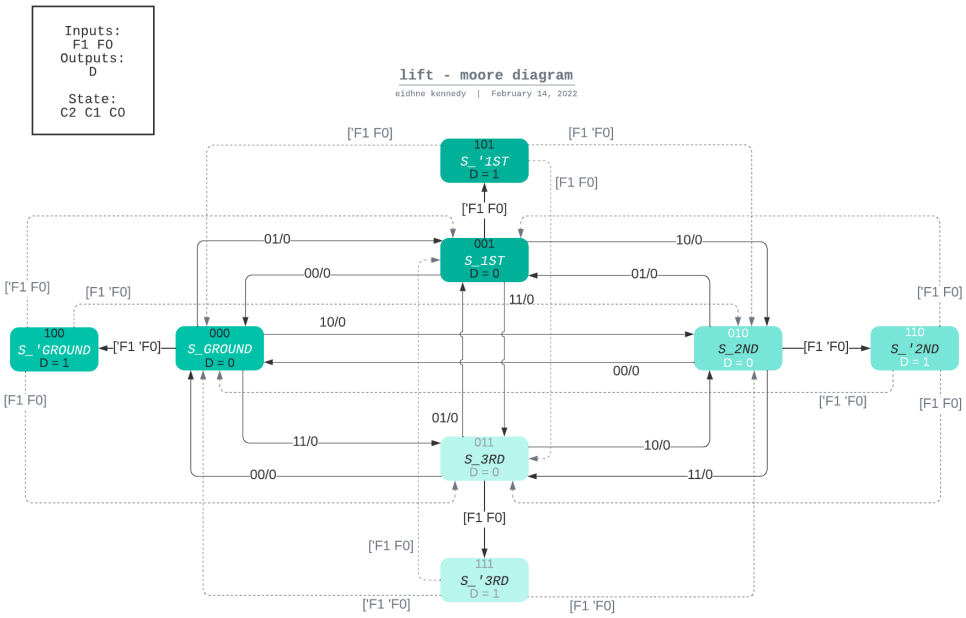
Lift - Moore design

state	$c_2c_1c_0$
ground	000
ground'	100
1st	001
1st'	101
2nd	010
2nd'	110
3rd	011
3rd'	111

input	f_1f_0
ground	00
1st	01
2nd	10
3rd	11

output	d
doors closed	0
doors open	1

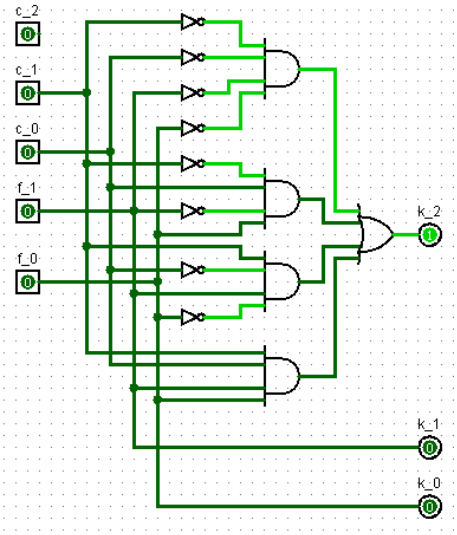
Already we can see that we need more states, and therefore more bits to encode them, as we can't have the self-loops in this design that were present in the previous one. To solve this problem, we have the prime states, ground', 1st', and so on, which will always output $d = 1$.



So, with more states, the next state logic will be more complex, however the output logic would be far simpler than for the Mealy implementation, as the boolean expression for d would just be $d = c_2$.

$c_2c_1c_0$	f_1f_1	$k_2k_1k_0$	d
000	00	100	0
000	01	001	0
000	10	010	0
000	11	011	0
001	00	000	0
001	01	101	0
001	10	010	0
001	11	011	0
010	00	000	0
010	01	001	0
010	10	110	0
010	11	011	0
011	00	000	0
011	01	001	0
011	10	010	0
011	11	111	0

$c_2c_1c_0$	f_1f_1	$k_2k_1k_0$	d
100	00	100	1
100	01	001	1
100	10	010	1
100	11	011	1
101	00	000	1
101	01	101	1
101	10	010	1
101	11	011	1
110	00	000	1
110	01	001	1
110	10	110	1
110	11	011	1
111	00	000	1
111	01	001	1
111	10	010	1
111	11	111	1



Moore next state logic

Comparison and contrast between Mealy and Moore designs

So, there are clearly major differences in implementation between Mealy and Moore designs. Below are some of the major ones in a table:

Mealy	**Moore**
Outputs are determined by inputs and current state	Outputs are only determined by current state
Outputs are on vertices in state diagram	Outputs are on states in state diagram
Less states needed	More states needed
Simpler next state logic	Simpler next state logic

Full finite state machine example

To better illustrate what a finite state machine is, and how it operates, let's look at a worked example. Consider an telescope, mounted on a moving platform, that looks at the stars. It can move in one of the eight cardinal directions, north, south, northwest, and so on. After moving, it checks the section of the sky to see if it can see light from a star. If so, it records that, and moves to another section of the sky. This process can be summarised as follows:

Algorithm 1 Telescope logic

```
while true do
    direction = input
    move(direction)
    if light == 1 then
        record = 1
    end if
end while
```

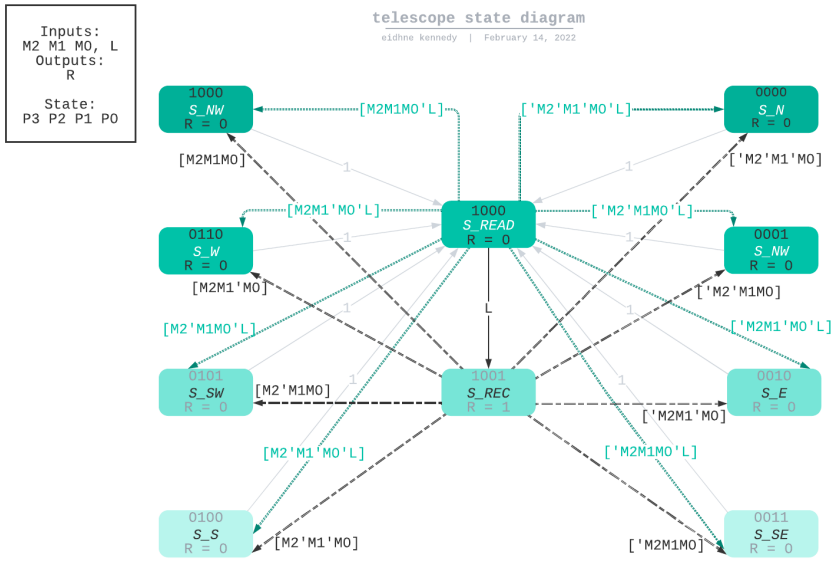
The approach I've chosen is a Moore machine.

state	$p_3p_2p_1p_0$
move north	0000
move northeast	0001
move east	0010
move southeast	0011
move south	0100
move southwest	0101
move west	0110
move northwest	0111
reading	1000
recording	1001
unused	1010-1111

input	variables
move direction	$m_2m_1m_0$
light	l

output	r
not recording	0
recording	1

Essentially, the machine is either moving or reading, and outputting $r = 0$, until $l = 1$ and it is in the reading state, 1000, at which it moves to state 1001, recording, where the output is 1. This process is outlined in the state diagram below.



Telescope state diagram

Not shown here are the dummy states 1010 through 1111, which all simply

move to the reading state, 1000. This has been factored into the state table and following logic, however.

$p_3p_2p_1p_0$	$m_2m_1m_0$	l	$v_3v_2v_1v_0$	r
0000	XXX	X	1000	0
0001	XXX	X	1000	0
0010	XXX	X	1000	0
0011	XXX	X	1000	0
0100	XXX	X	1000	0
0101	XXX	X	1000	0
0110	XXX	X	1000	0
0111	XXX	X	1000	0
1000	000	0	0000	0
1000	001	0	0001	0
1000	010	0	0010	0
1000	011	0	0011	0
1000	100	0	0100	0
1000	101	0	0101	0
1000	110	0	0110	0
1000	111	0	0111	0
1000	XXX	1	1001	0
1001	000	X	0000	1
1001	001	X	0001	1
1001	010	X	0010	1
1001	011	X	0011	1
1001	100	X	0100	1
1001	101	X	0101	1
1001	110	X	0110	1
1001	111	X	0111	1
1010	XXX	X	1000	0
1011	XXX	X	1000	0
1100	XXX	X	1000	0
1101	XXX	X	1000	0
1110	XXX	X	1000	0
1111	XXX	X	1000	0

Figure 1: v_3 karnaugh map

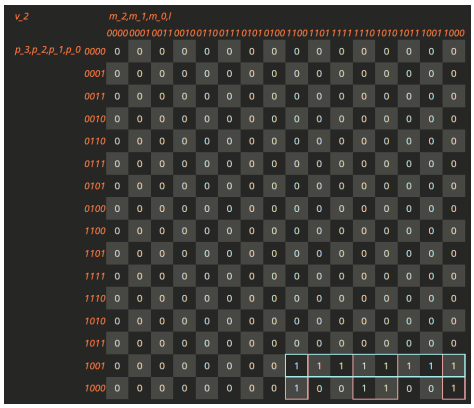


Figure 2: v_2 karnaugh map

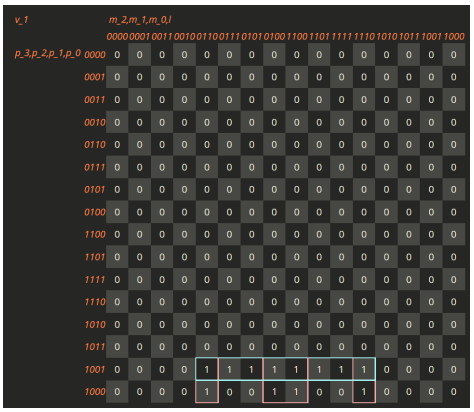


Figure 3: v_1 karnaugh map

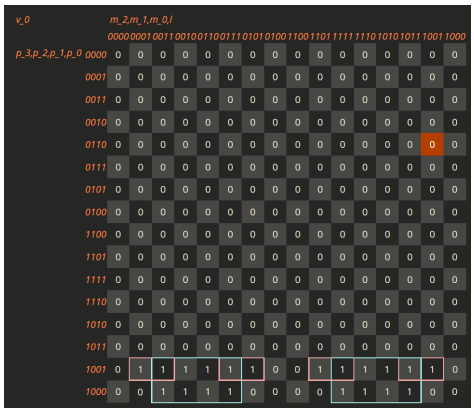


Figure 4: v_0 karnaugh map

(images created using this website [10])

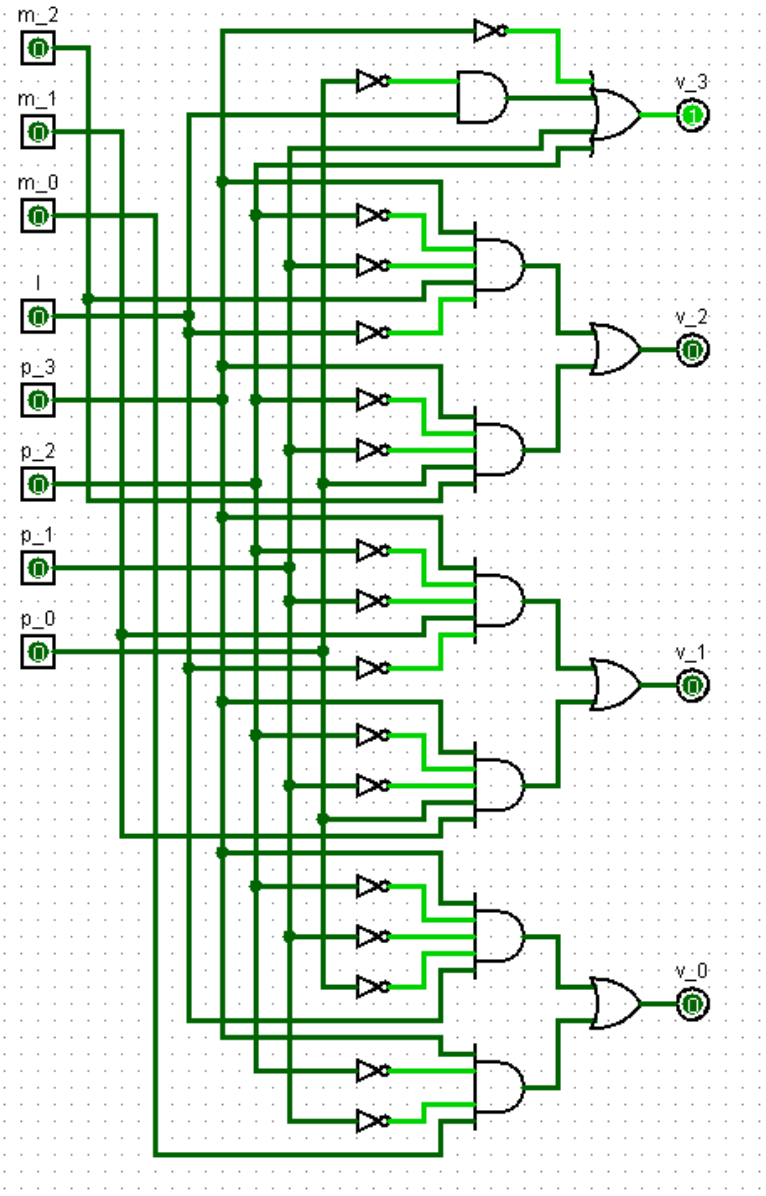
$$v_3 = \overline{p_3} + \overline{p_0}l + p_1 + p_2$$

$$v_2 = p_3\overline{p_2}\overline{p_1}m_2\overline{l} + p_3\overline{p_2}\overline{p_1}p_0m_2$$

$$v_1 = p_3\overline{p_2}\overline{p_1}m_1\overline{l} + p_3\overline{p_2}\overline{p_1}p_0m_1$$

$$v_0 = p_3\overline{p_2}\overline{p_1}\overline{p_0}l + p_3\overline{p_2}\overline{p_1}m_0$$

Other resources used, but not specifically referenced in this report: [11], [12]



Telescope next state logic

References

[1] J. Wang and W. Tepfenhart, "Formal methods in computer science," 2019.

[2] "Automata theory, https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html."

[3] W. Rudin, *Principles of Mathematical Analysis*. McGraw-Hill u.a., 1964.

[4] R. Munroe, "Wikipedian protester."

[5] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity (1943)," *Ideas That Created the Future*, p. 79–88, 2021.

[6] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, p. 1045–1079, 1955.

[7] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata Studies. (AM-34)*, p. 129–154, 1956.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers principles, techniques and tools*. Addison-Wesley, 1985.

[9] "Intelligent diagramming, www.lucidchart.com."

[10] "karnaugh map solver - www.charlie-coleman.com/experiments/kmap/."

[11] G. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and logic*. Cambridge University Press, 2010.

[12] "Generating finite state machines from ... - https://www.microsoft.com/en-us/research/wp-content/uploads/2002/07/issta02.pdf."