CT-101 Assignment 4 – Finite State Machines – Liam Holland – 21386331

Part 1 – What is a Finite State Machine?

In this first section of the report, I intend to cover in detail the purpose and applications of finite state machines (FSMs) and go over their origin in computer science. The FSM is today the simplest of all its family of computing system models, but it can still be employed to describe a wide range of systems.

Definition

A finite state machine is a behavioural model of a hypothetical system, in which only one out of a finite number of states can be active at any given time. They produce an output after receiving an input and considering the current state that the machine is in. It is possible to then implement these designs using hardware or software in order to simulate sequential logic. (1) (2) (3) (4)

History

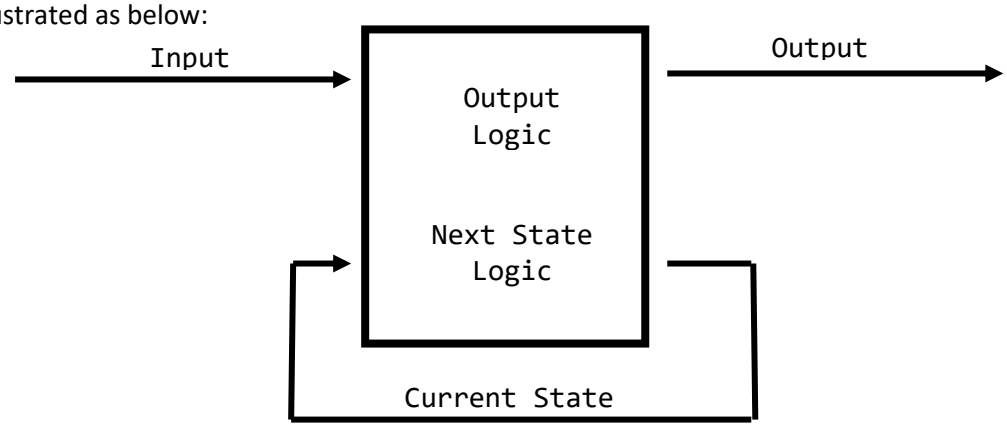
As can be inferred from the above definition, the FSM has a wide range of applications across many different areas. For example, they are used in, of course, computer science, but also in electrical engineering, mathematics, biology, linguistics and even in philosophy. In fact, this wide range of applications goes all the way back to their inception, when they were originally devised as a means of mapping the human mind. This is partly due to the fact that in 1943, the year Warren McColloch and Walter Pitts first described such a machine, computer science was not a widely studied, or even considered field. McColloch and Pitts were neurophysiologists. It was over 10 years before G.H Mealy and E.F Moore would describe their own versions of FSMs in relation to computer science. (5)

The FSM stems from a branch of computer science known as automata theory. This family of machines includes the famous Turing Machine, or as we know them today, computers. However, the Turing machine, conceived by Alan Turing in 1937, is, in fact, a model of an *infinite* state machine. It was intended to be programmable to any problem, thus having an infinite number of states. This is, realistically, impossible. But as a finite state machine can have a number of states that is equivalent to 2 to the power of the total number of available bits of memory, this is essentially infinite in modern computers. So, while they are *technically* FSMs, there could also be an infinite number of interactions between the different bits. Therefore, despite having finite memory, they are modelled as Turing Machines. (3) (5)

It has even been argued that, because FSMs react to a change of input by changing the output based on the previous state, or the history of inputs, they have sufficient capabilities to map a human, as originally desired. In other words, some believe that humans are finite state machines. (3)

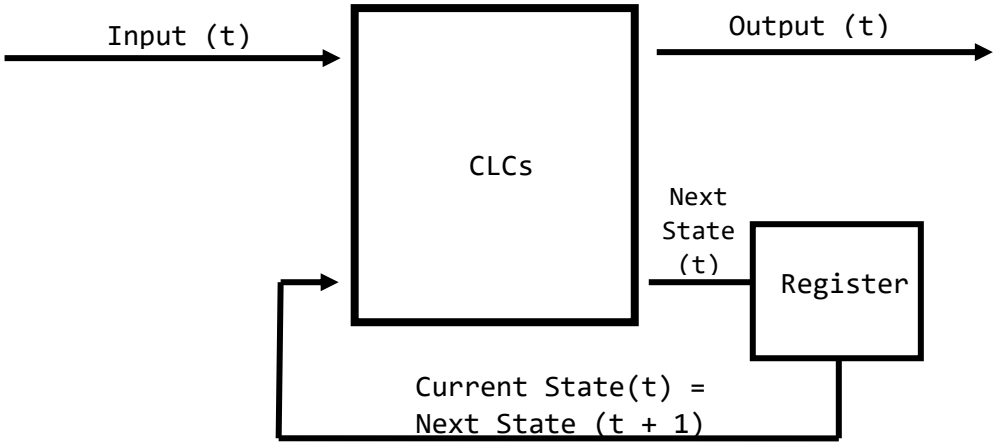
The Machine

So, what does the actual machine look like, in theory? Well, as described above, the objective of the FSM is to produce an output based on a change in input and the current state that the machine is in, the output being in two parts, an output and the next state. Thus, a Finite State machine could be illustrated as below:



As you can see, the FSM contains logic for both the output and the next state of the machine. The calculated next state, based on input and the current state, is passed back into the machine as the new current state, while the input either remains the same or changes. This allows the machine to remain in the same state for as long as the input stays the same, so long as that is desired by the designer.

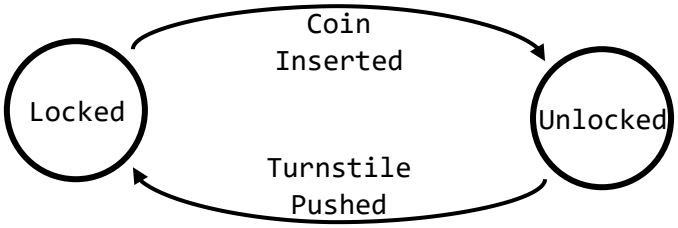
However, while this diagram conveys the basic idea behind FSMs, it fails to truly display the way they actually work in practice. Computers run on a clock cycle, and thus, FSMs do as well. Therefore, an extra component must be added to the system in order to allow the machine to *remember* the next state, which it calculates on one cycle and passes back into the machine on the next. Therefore, with these changes made, the diagram will look something like this:



Here, everything operates on the clock cycle, t. The register takes one clock cycle, and passes the next state out on the next, where it represents the current state on the new clock cycle. It would also be possible to pass the output through a register, delaying the output by one clock cycle but also making it available for other calculations. This would be necessary in a system in which you are stringing multiple FSMs together.

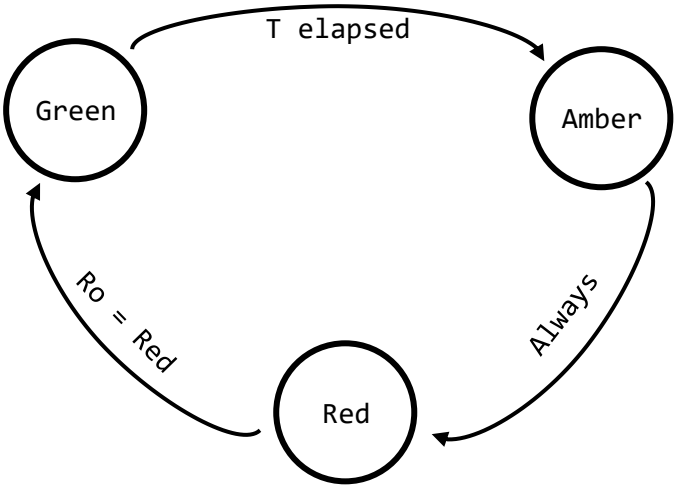
Examples

1. A coin operated turnstile:
   The FSM in this case only has two states, making it simple and easy to understand. These states are, intuitively, locked and unlocked. The machine will begin in the locked state. Putting the correct coin in the slot will cause it to transition to the unlocked state and pushing the arm of the turnstile around will transition it back to the locked state. Simple, yet effective.

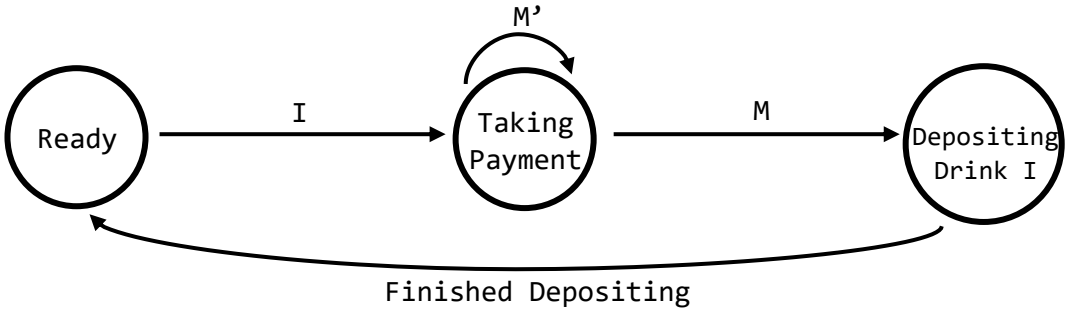


2. Traffic Lights:
   This is a classic example of an FSM. Consider traffic lights at a crossroads. Say that opposite lights are connected, creating two roads. The lights on these roads would be connected so

that after a certain time T has elapsed, the lights would go from Red to Green on one road, after going from green to amber and amber to red on the other. The states here would be the colour of the lights, and the transitions would depend on T and whether the other road (Ro) had red lights. Both roads could also use the same FSM in this case, as Ro could just represent the other lights.



3. A vending machine:
This FSM would have multiple layers of operation and depend heavily on user input. The states of this machine would be Ready, Taking Payment and Depositing Drink. The machine would go from one to the other based on the user's input; first based on what button they pressed and then based on how much money they had inserted. I will represent this in a diagram simpler than one you would use to actually design the machine. In reality, you would have many states, ranging from the certain shelf to operate to the amount of money the user had inserted. In my diagram, I will use the variable I (user input – I also represents the drink selected) and M (tracks if the correct amount of money has been inserted).

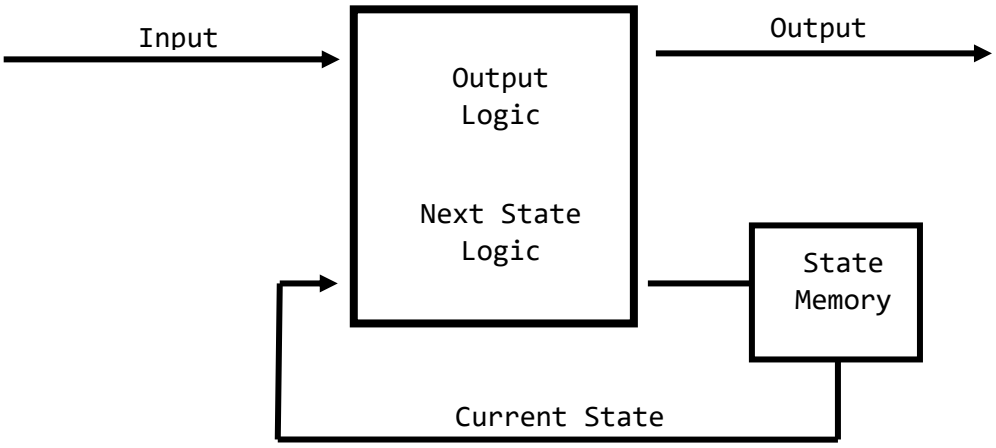


Part 2 – Mealy vs. Moore

Underneath the umbrella of finite state machines are two types of sub-machines. These are the Mealy machine and the Moore machine. These machines are generalised and far more powerful designs which build upon the earlier theory. They were written about in two separate papers by George H. Mealy and Edward F. Moore in 1955 and 1956 respectively. There are a multitude of differences between the two methods of modelling FSMs, but each have their advantages and disadvantages. (5)

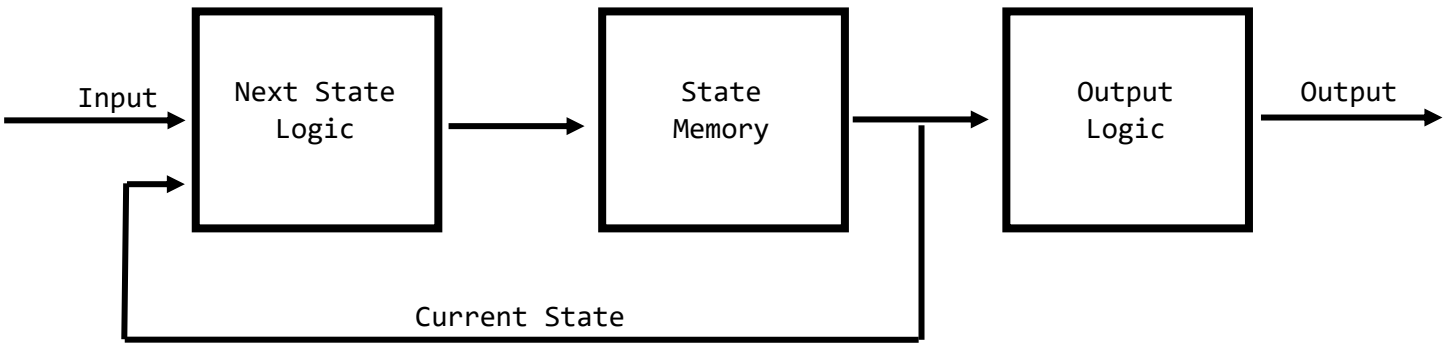
First, the Mealy machine. This type of FSM uses both the input and the current state as inputs which are used to calculate the output. It produces the output asynchronously, meaning that the output can change immediately upon a change in the input occurring. While this may seem advantageous, it can also cause problems in some situations. For example, if you wanted to chain two or more Mealy machines together, you could cause asynchronous feedback, causing timing errors and most likely transitioning to an undefined state. (6) (7)

A diagram of a Mealy FSM looks like the following:



The Moore machine, while it will produce the same result as the Mealy machine, will do it slightly differently. First of all, it will return the output one cycle later than the Mealy, as this design is synchronous with the clock edge. This makes Moore machines far more stable than the Mealy machine as well. The drawback of this is that it generally takes more states to represent the same solution as a Moore machine. This gives rise to a handy method of differentiating the two machines, by remembering that the *Moore* machine usually has *more* states. (8)

The output of the Moore machine is based solely on the current state. The input of the machine is used to calculate the next state, which is passed into the output logic as the current state and back into the next state logic. The diagram looks something like this:



Comparison Table

Now I will go over advantages and disadvantages of the two machines in the form of a comparison table.

	Mealy	Moore
Input	Input to the machine is the input and also the current state of the machine. Input is used directly in producing the output	Input is simply the input to the machine. Input is only used indirectly in producing the output
Output	Depends on the input and the current state. Calculated on the same clock cycle as the input. The output is based on the transition	Depends only on the current state. Calculated on the next clock cycle to the input. Output is based on the state
Stability	Can be unstable, as output is produced asynchronously. Problems and errors can arise if the input is variable	Very stable as the output is produced on a different clock cycle and the output depends only on the current state
Speed	Faster thanks to the output being immediate	Slower due to the clock edge timing and the fact that the next state logic goes through the register before the output is calculated
Modelling	The model of a Mealy machine is often more complex but also more compact as it requires less states to be visualised	The model of a Moore machine can be easier to understand but is generally far larger due to the increased number of states

As you can see the advantages of using one type of machine over the other depends entirely on the situation you are using them in. The speed of the Mealy is desirable, yet it comes at the cost of stability, while the inverse is true for Moore machines. This has led to research being done into transforming a Mealy machine into a Moore machine, a process which allows the designer to take advantage of all the benefits of both machines. (7)

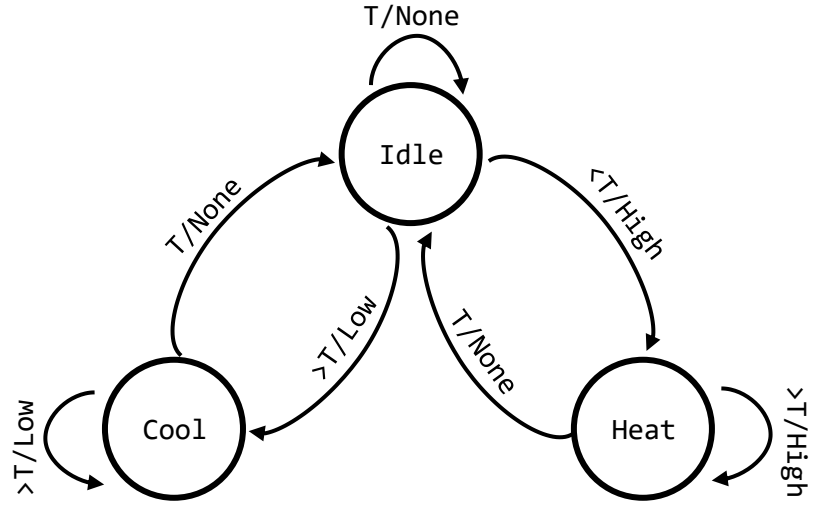
Example

Finally, I will give an example of an air conditioner system represented in both Mealy and Moore state diagrams. The system we are mapping will maintain the temperature in a room within the range T; heating if it drops below and cooling if it rises above. It will produce an output O will return High, Low or None for power usage, High being used for heating and Low for cooling.

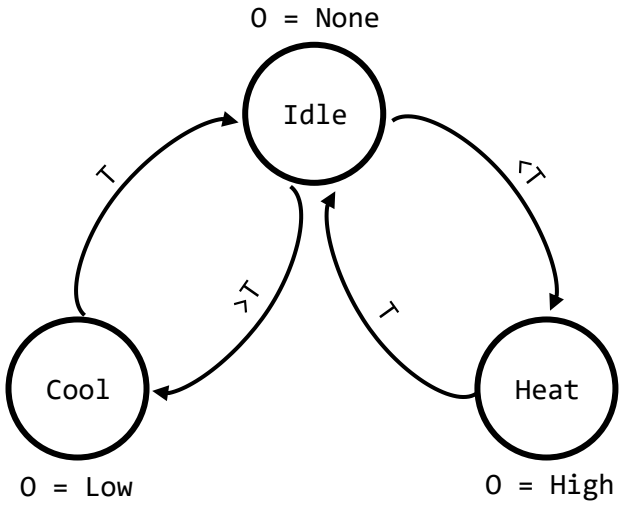
As this is a simple machine, the state diagram for both the Mealy and Moore machines will be identical:

Current State	Temperature	Next State	Output
Idle	T	Idle	None
Idle	<T	Heat	High
Idle	>T	Cool	Low
Heat	T	Idle	None
Heat	<T	Heat	High
Cool	T	Idle	None
Cool	>T	Cool	Low

Next will be the state diagrams, starting with the Mealy implementation:



And now for the Moore version:



As you can see, the Moore version of the diagram is far simpler in this case, and probably the better choice for a method to represent this system. This is largely due to the outputs being associated with the states rather than the self-transitions, so they can then be missing in the Moore diagram.

Part 3 – Full FSM Example

For this section of the assignment, I have chosen to create an example of a character animation controller for a video game. This is an example which is both easy to interpret into a real-world system, as well as being closely related to computing.

The purpose of this finite state machine is to change the animation that a 2d character has applied to it based on the state of said character in the game. Due to the fact that games rely heavily on input from the user and quick feedback, I believe that a Mealy machine is the appropriate implementation to use for this FSM.

Let's say that the player character is capable of the following actions:

- Jumping
- Running
- Ducking
- Shooting

These four actions will take the role of the four possible inputs the user can provide. In order to keep these inputs simple, they will be encoded using 2 bits, meaning that the first input – Jump – will be encoded as 00. To avoid this appearing as having no input, and also to allow me to define what happens when an input is held or released, I will have a second, 1 bit, input called P, which will be 1 when a button is pressed and 0 when it is released. This means that I can use the input 00, along with the encoded states, to represent a simpler and easier to interpret logic for this FSM.

Next, we have to define the rules of movement in the game:

- The player can only jump or run when they are on the ground
- The player cannot do any other actions apart from shoot while falling or ducking
- The player can do no other actions while jumping
- If the player ducks while running, they will slide
- If there is no input, the player character will be idle
- The player cannot shoot while sliding
- If the player does not release the duck button before the slide ends, they will remain in the ducking state after the slide ends

With these rules in place, we can define the following states for the player character:

- Falling
- Sliding
- Idle
- Falling and Shooting
- Running and Shooting
- Ducking and Shooting

Before presenting the state diagram, I will first label each of the states, inputs and outputs so that the table is more concise.

States:

S_0 = Idle

S_1 = Jumping

S_2 = Running

S_3 = Ducking

S_4 = Sliding

S_5 = Shooting

S_6 = Falling

S_7 = Falling and Shooting

S_8 = Running and Shooting

S_9 = Ducking and Shooting

Inputs (I):

0 = Jump

1 = Run

2 = Duck

3 = Shoot

Inputs (P):

1 = Input / Input Held

0 = No Input / Input Released

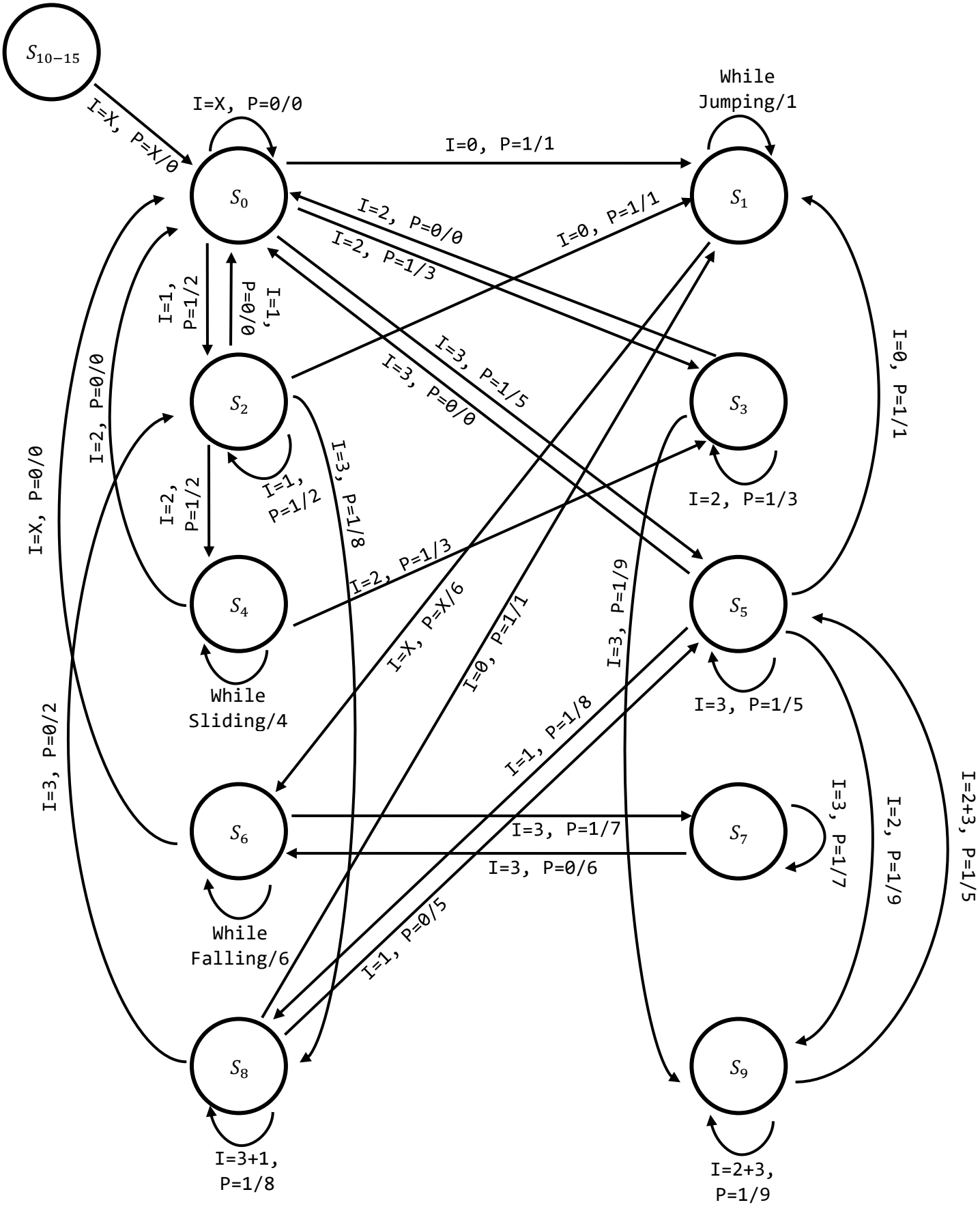
Outputs (O):

Numbered 0 through 9 to represent 10 animation files associated with the state the FSM is transitioning to – The next state.

Here is the state table. Note that the self-transitions for states 1, 4, 6, 8, and 9 are special cases. Due to the limitations of the variables I have used in this FSM, they cannot be represented properly in the encoded state table or the next state logic, but will be included in the state diagram as they appear here in the state table. The states are also colour coded for further clarity.

State	I	P	Next State	Output
S0	X	0	s0	0
	0	1	s1	1
	1	1	s2	2
	2	1	s3	3
	3	1	s5	5
s1	X	X	s6	6
	While Jumping		s1	1
s2	1	0	s0	0
	1	1	s2	2
	0	1	s1	1
	2	1	s4	4
	3	1	s8	8
s3	2	0	s0	0
	2	1	s3	3
	3	1	s9	9
s4	2	0	s0	0
	2	1	s3	3
	While Sliding		s4	4
s5	3	0	s0	0
	3	1	s5	5
	0	1	s1	1
	1	1	s8	8
	2	1	s9	9
s6	X	0	s0	0
	3	1	s7	7
	While Falling		s6	6
s7	3	0	s6	6
	3	1	s7	5
s8	3	0	s2	2
	1	0	s5	5
	0	1	s1	1
	3 + 1	1	s8	8
s9	2	0	s5	5
	3	0	s3	3
	2 + 3	1	s9	9
s10 -> s15	X	X	s0	0

The following is the state diagram for the FSM:



For the encoded state table, I swapped the position of the Current State column to be third, after the I and P input columns. This was to make the Karnaugh maps more manageable, as they were already complex enough with 7 variable bits. Ordering the binary this way allowed me to have all 4 bits for the state on the top of the K-map, increasing its readability greatly. As previously stated, the special case self-transitions cannot be included in this table.

I0I1	P	S0S1S2S3	N0N1N2N3	Output
XX	0	0000	0000	0
00	1	0000	0001	1
01	1	0000	0010	2
10	1	0000	0011	3
11	1	0000	0101	5
XX	X	0001	0110	6
01	0	0010	0000	0
01	1	0010	0010	2
00	1	0010	0001	1
10	1	0010	0100	4
11	1	0010	1000	8
10	0	0011	0000	0
10	1	0011	0011	3
11	1	0011	1001	9
10	0	0100	0000	0
10	1	0100	0011	3
11	0	0101	0000	0
11	1	0101	0101	5
00	1	0101	0001	1
01	1	0101	1000	8
10	1	0101	1001	9
XX	0	0110	0000	0
11	1	0110	0111	7
11	0	0111	0110	6
11	1	0111	0111	5
11	0	1000	0010	2
01	0	1000	0101	5
00	1	1000	0001	1
10	0	1001	0101	5
11	0	1001	0011	3
X	X	1010 -> 11	0000	0

Using this data, I was able to construct 4 Karnaugh maps and derive an equation for each of the 4 bits N0, N1, N2 and N3.

N0:

I0I1P\S0S1S2S3	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
111	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

N0(I0, I1, P, S0, S1, S2, S3) = I0'I1PS0'S1S2'S3 + I0I1'PS0'S1S2'S3 + I0I1PS0'S1'S2

N1:

I0I1P\S0S1S2S3	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
011	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
110	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
111	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
101	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
100	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

N1(I0, I1, P, S0, S1, S2, S3) = S0'S1'S2'S3 + I0'I1P'S0S1'S2'S3' + I0I1'P'S1'S2'S3 + I0I1'PS0'S1'S2S3' + I0I1S0'S1S2S3 + I0I1PS0'S1'S2' + I0I1PS0'S1S2 + I0I1PS0'S2'S3

N2:

I0I1P\S0S1S2S3	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
011	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
010	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
110	0	1	0	0		1	0	0	0	0	0	0	0	0	1	1
111	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
101	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
100	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

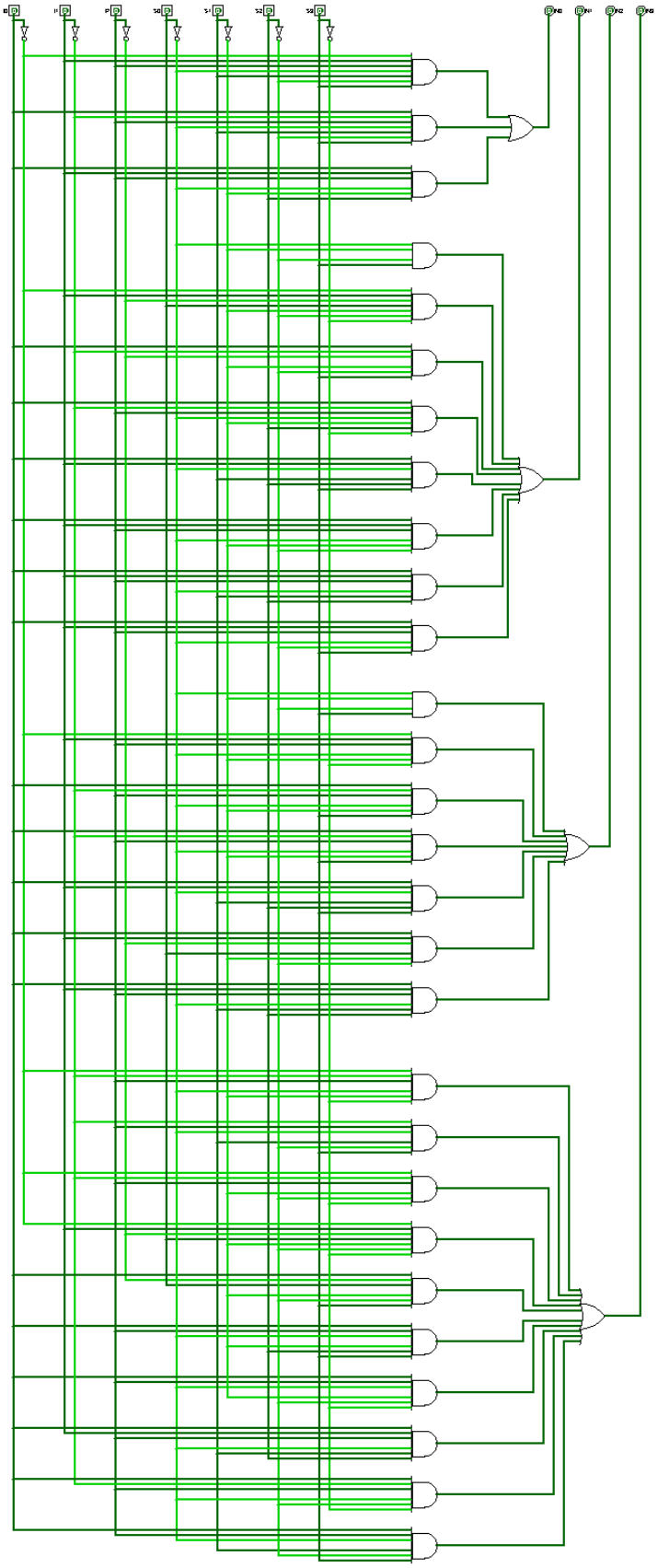
N2(I0, I1, P, S0, S1, S2, S3) = S0'S1'S2'S3 + I0'I1PS0'S1'S3' + I0I1'PS0'S1'S3 + I0I1'PS0'S1'S3 + I0I1S0'S1S2S3 + I0I1P'S0S1'S2' + I0I1PS0'S1S2

N3:

I0I1P\S0S1S2S3	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
001	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1
011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
111	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0
101	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

N3(I0, I1, P, S0, S1, S2, S3) = I0'I1'PS0'S1'S3' + I1'PS0'S1S2'S3 + I0'I1'PS1'S2'S3' + I0'I1P'S0S1'S2'S3' + I0P'S0S1'S2'S3 + I0PS0'S1'S2S3 + I0PS0'S1'S2'S3' + I0I1PS0'S1S2 + I0I1'PS0'S2'S3' + I0PS0'S1S2'S3

Using these 4 equations, I was able to construct the combinational logic for the next state logic of the FSM.



References

1. **What is a State Machine?** ***itemis.com.*** **[Online] https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines.**

2. Finite State Machines. ***Brilliant.org.*** **[Online] https://brilliant.org/wiki/finite-state-machines/.**

3. Who Invented the Finite State Machine? ***moviecultists.com.*** **[Online] https://moviecultists.com/who-invented-finite-state-machine.**

4. What is a Finite State Machine? ***medium.com.*** **[Online] https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c.**

5. Automata Theory. ***cs.stanford.edu.*** **[Online] https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html.**

6. Async. vs. Sync. [Online] https://jjm.staff.sdu.dk/MMMI/Logic/Statemachines/Asyncronous_Vs_Synchronous_FSM/index.htm .

7. Mealy-to-Moore Transformation. ***IEEE Xplore.*** **[Online] https://ieeexplore.ieee.org/abstract/document/7934561.**

8. Moore Machine. ***Science DIrect.*** **[Online] https://www.sciencedirect.com/topics/computer-science/moore-machine.**