Automata Theory: A Look into Finite-state Machines

Lydia Csaszar
Advisor: Dr. Mary Porter

November 29, 2022

Contents

1 Introduction

Around the turn of the 20th century, in the study of computational theory, a new branch of theoretical computer science was established. Both mathematicians and those who would become the modern-day equivalent of computer scientists were focused on developing machines that could imitate certain life features. The goal of their work was to create machines that would complete computations more quickly and reliably than the machines of the day. Thus, automata theory was founded to discover the logic of computation concerning simple machines called automata. It is because of Automata theory that we as a society can understand how machines' computer mathematical function and solve problems given to them. We can think of Finite-state automatons as abstract models of machines that perform a certain computation when given a particular input by moving through a series of states. There are four main classifications of automata; finite-state machines, pushdown automata, linear-bounded automata, and turing machines [11]. For the purposes of this paper, we will be exploring finite-state machines. Finite-state machines and the subclass of Deterministic Finite-state Machines, also known as Deterministic Finite-state Automata (DFAs), are pivotal to Automata Theory as they are the class of computational models made up of theoretical states that store the current states the machine is at and will change between states based on a given input [7].

2 Finite-state Automata

In this section we will work to develop an understanding of what Finite-state Automata are. Before we jump into understanding what a Deterministic Finite-state Automata (DFA) is, we must lay the groundwork to understand the inputs of a DFA. Regular expressions and Languages are part of the input process which defines whether or not we can expect the machine to work and under what conditions it can work. Therefore the next two subsections will cover the necessary information and theorems needed to better understand Finite-state Machines.

2.1 Regular Expressions

Regular expressions, or REs for short, are a type of expression via language patterns. These expressions "algebraically" specify a language by starting with a base alphabet Σ and building to a language L using a set of fixed operations [9].

The four fundamental algebraic properties that Regular expressions follow are the following, where each letter represents, and single regular expression [5]. Note that the vertical bar in this context is to reference alternation, which in computer science terms makes this the pipe symbol that represents or in a regex engine.

- Associativity: $a|(b|c) = (a|b)|c$
- Distribution: $a(b|c) = ab|ac$
- Commutativity: $a|b = b|a$
- Idempotency: $a** = a*$

We can then begin our understanding of building from the base alphabet Σ by looking at the definition of regular expressions over Σ.

Definition 2.1. The set of regular expressions over alphabet Σ is defined inductively in this way:

1. The symbols "$\emptyset$" and "ϵ" are regular expressions.
2. The symbol "$\mathbf{x}$" is a regular expression if $x \in \Sigma$
   (Author's Note: The boldface is used to distinguish the symbol $\mathbf{x}$ form the element $x \in \Sigma$)
3. If A and B are regular expressions, then so is $A|B$ (the alternation of A and B), and AB (the concatenation of A and B), and A^* (The Kleene Star of A).

2.2 Kleene Star and Languages

Stephen Cole Kleene was the first to formally define REs as an element of automata theory and computing [10]. His work led to regular expressions being able to be interpreted in different forms across programming languages like Perl, text editors like vim, and command line tools like grep [5]. Kleene developed a set of inductive rules of precedence by understanding the relationship between the strings in the alphabet and the language of those strings.

Definition 2.2. A **regular expression** s is a string that denotes L(s), which is a set of strings drawn from an alphabet Σ. L(s) is subsequently known as the "Language of s" [5].

L(s) is defined inductively with the following base cases:

- If a $\in \Sigma$ then a is a regular expression and L(a) = {a}.
- ϵ is a regular expression and L(ϵ) contains only the empty string.

Then, for any regular expression s and any regular expression t:

1. $s|t$ is a regular expression such that $L(s|t) = L(s) \cup L(t)$.
2. st is a regular expression such that L(st) contains all strings formed by the concatenation of a string in L(s) followed by a string in L(t).
3. s^* is a regular expression such that L(s^*) is L(s) concatenated zero or more times.

The operation in rule number three, which is the Kleene star also known as Kleene Closure, has the highest precedence in the order of operations. In contrast, the operation in rule number three has the lowest precedence and is known as alternation. The operation in rule number two is our working definition of concatenation. Since the rules have a given order of operation, from that given regex pattern the machine can then search to find any string regardless of the programming language [5].

Another important idea related to the Kleene star and regular expressions is interpreting the sequence of symbols within a regular expression. One common operator is the "+"; for example, $X+$ would be interpreted by a machine as XX^*. This abbreviation for the Kleene star is often referred to as syntactic sugar by computer scientists and mathematicians because it is a valid shorthand that is understood by both humans and machines.

Definition 2.3. Given a regular expression R over an alphabet Σ, we inductively define L(R), the language specified by R, as follows:

1. $L(\emptyset) = \emptyset$ which is the empty set
2. $L(\epsilon) = \{\epsilon\}$, which is the set whose only element is the empty string, ϵ.
3. $L(\mathbf{x}) = \{x\}$, which is set whose only element is the one-character string "x".
4. $L(AB) = L(A)L(B) = \{ab \mid a \in L(A), b \in L(B)\}$.
5. $L(A|B) = L(A) \cup L(B)$
6. $L(A^*) = L(A)^* = \{\epsilon\} \cup \{a_1a_2 \ldots a_n \mid n \in \mathbb{N}, a_i \in L(A)\}$. Note that this is a natural extension of the Kleene star notation defined above in Definition 2.2 [9].

We will now define what it means for a language to be regular.

Definition 2.4. A language L over alphabet Σ is **regular** if there exists a regular expression R over the alphabet Σ such that $L = L(R)$ [9].

The following theorem, called Kleene's Theorem, concerns the definition above. However the proof of this theorem is beyond the scope of this paper.

Theorem 2.1 (Kleene's Theorem). *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

2.3 Deterministic Finite-state Automata

A deterministic finite automaton, DFA for short, is characterized by a 5 component system. DFAs are a particular case of Finite-state Automata where every state has no more than one outgoing path for a given state. There is no ambiguity in DFAs; for every combination of state and input symbol, there is precisely one choice that matches the next move. The relation to Finite-state automata means the movement within the machine is wholly determined by the input string into the machine. Given this information, we would then have the following 5 components.

1. An arbitrary finite set, denoted by Σ, called the input alphabet.
2. Secondly, another arbitrary finite set Q where each element is called a state.
3. Next, we have the transition function denoted by δ, which determines which state the machine will move to from the current state. This function δ maps each pair (a state from Q, an input from Σ) to a state from Q.
4. A start state s_0 where $s_0 \in Q$.
5. We must also list A, the set of accepting states, where $A \subseteq Q$.

Deterministic Finite-state Automata operate on an input string given to the machine. The start state s then reads the input string characters one-by-one till it reaches the end of the string [9]. Each character read from the string determines the transition to the next state by the machine applying that character to the transition function δ. By the end of the string, the DFA determines if it ends in the accepting state. If it does end in the accepting state, then the string x is accepted.

There is a key limitation to DFAs, which can make them unsuitable for certain use cases, and therefore the reader should take care not to use them. DFAs are not good at counting, which is asserted in the following lemma.

Lemma 2.2. *Let c be a constant and $L = \{1^c\}$, the singleton set that contains a string of c many 1s. Then no DFA with fewer than c states can accept L [9].*

Proof by Contradiction . Assume that some DFA called M with less than x states accepts L. Let the sequence of $s_0, s_1, s_2, \ldots, s_c$ be the states that are traversed by the DFA M to accept the string 1^c (and so $s_c \in A$ is an accepting state). By the pigeonhole principle, [8], some state is repeated twice, $s^* = s_i = s_j$ with $j > i$. Let $t = j - i$. Therefore the DFA M must also accept the string $1^{(c+t)}, 1^{(c+2t)}$, etc; the behavior of M over these inputs is as follows. Starting from s_0, after reading 1^i, M ends up in the state $s^* = s_i$. From this point forward, for every t many 1s in the input, M will loop back to the state $s^* = s_i = s_j$. After sufficiently many loops, M will read the final $c-i-t$ many 1s, transitioning from state $s^* = s_j$ to s_c, an accept state. Thus we have a contradiction since the DFA M accepts infinitely more strings than the language L [9]. $\square$

2.4 Non-Deterministic Finite-state Automata

A Nondeterministic Finite-state Automaton, or NFA for short, is a valid finite automaton but contains some ambiguity from an extra transition called ϵ that makes it more difficult. This ϵ transition is the empty string [5]. This transition can be taken without consuming and inputting symbols. Since there are no symbols for the machine to determine the path, we assume that the ambiguity can be interpreted in one of two ways.

- Ambiguity via crystal ball: This interpretation suggests that the NFA ”knows” what the best choice is by some means outside the NFA itself. The crystal ball interpretation only works in a theoretical simulation but not in a physical setting [5].
- Ambiguity via many worlds: This interpretation suggests that the DFA exists in all allowable states simultaneously. Therefore the machine will run to completion, and if it ends in any of the accepting states, the NFA has accepted the input successfully [5].

3 Common Case Example: A Light Bulb

One common example is a plugged-in lamp with a working light bulb. Let us first consider a light bulb with an off button and an on button, like in Figure 1. This plugged-in lamp is expected to

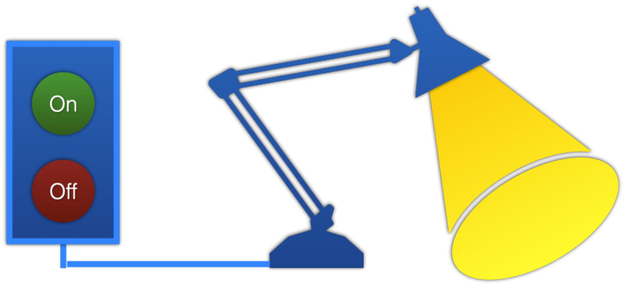


Figure 1: Basic Lamp with off/on buttons

light up when we press the on button. When we press the off button, the gate in the wire closes, so the connection between the electrical wires and the bulb connects, so we have light. Similarly, the gate opens when we press the off button, and the connection is lost, so the bulb turns off. The following mathematical characteristics would describe the result.

1. Σ = {On,Off}

2. Q = {Light,Dark}

3. δ(Light,On) = Light, δ(Light,Off) = Dark, δ(Dark,On) = On, δ(Dark, Off) = Dark

4. The start state is $Dark \in Q$

5. The set of accepting states is $A = \{Dark, Light\} \subseteq Q$

We note that our transition states are in the set Σ and are the values of On and Off. We define our states as Light or Dark, representing whether the bulb is lit. We have our delta function, which maps every possible combination of state and transition values to a correlated state value. Then we pick the start state to be Dark because we often walk into a room and turn on a lamp instead of turning it off, but the choice in this state machine is inconsequential to the machine's functionality. Lastly, we note that both possible states are accepting states in this machine. This translates nicely into a Finite-state Machine Diagram shown in the figure below [2][3].

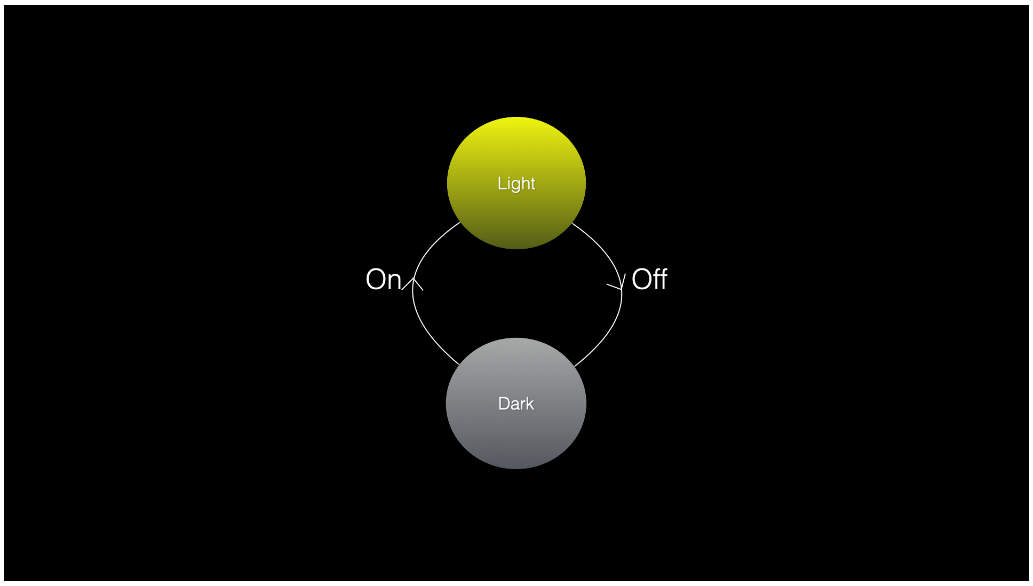


Figure 2: Finite-state Machine Diagram of a Light Bulb

We are fortunate that this is a simplistic Finite-state Machine, but we can verify this by creating our Karnaugh map of simplification like in Figure 3.

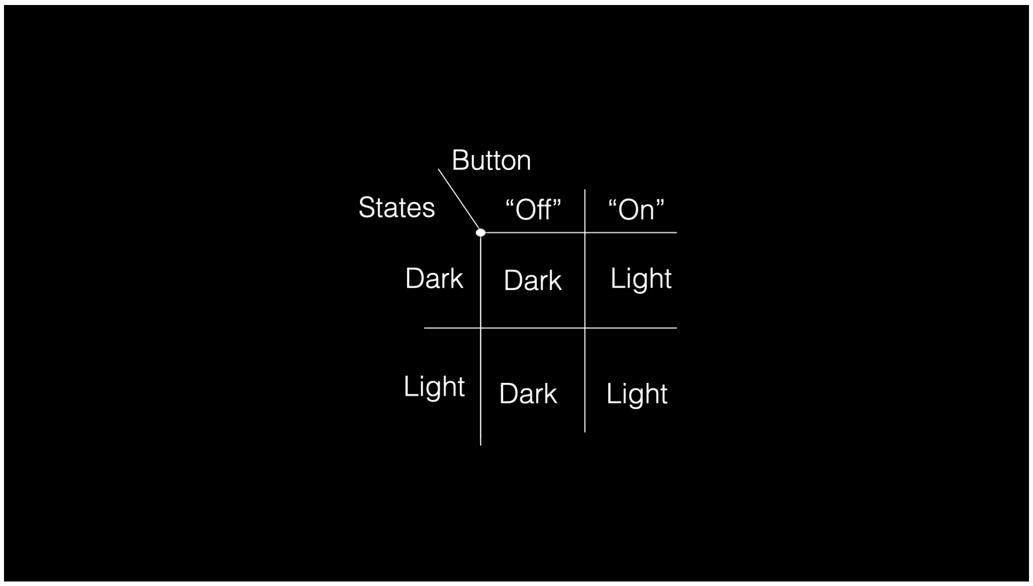

Button
States
"Off"
"On"
Dark
Dark
Light
Light
Dark
Light

Figure 3: Karnaugh Map of a Finite-state Machine of a Light Bulb

We see these combinations work with mapping the delta functions from above, so we have created a Finite-state Machine with minimal states.

4 Complex Case Example: Traffic Light

Let us consider a more complex example of lights. Consider we have an inner section of two roads. Hanging in the center of inner sections above the roads is a single-stop light facing both streets. We know that a single stop light will have the lights on opposite sides of the light synced to the same movement of Red, Yellow, and Green. Let's try to conceptualize the possible pairs of lights for each side of the light. It is easier to start thinking about the pairs that would only be caused by a severe error or reset. For instance, when a power outage occurs, you might see stop lights go to all red to signal a mistake, and you should treat it as a 4-way stop. We would also consider that the combinations of green and green or yellow and yellow would be a significant error that you would want to go to red and red so that the light can be reset. You also would not have the combination of green and yellow or yellow and green since you would not want all traffic moving simultaneously. That would then leave us with the final combinations we would regularly see when driving, which are green and red or red and green and the combo of yellow and red or red and yellow. We could initially assume that we would want one street to have a sensor waiting for a car to stop before transitioning between states with some delay. We can then define our mathematical characteristics of the states.

1. Σ = {Car,Delay}
2. Q = {GG, RR, YY, GY, YG, RG, GR, YR, RY}
3. δ: $Q \times \Sigma \rightarrow Q$, where δ is defined as in the figure for this Finite-State Machine.
4. The starting state is $RR \in Q$
5. The set of accepting states is $A = \{GR, YR\} \subseteq Q$

This would then translate to a Finite-state Diagram like figure 4.

As we can see, there is some issue defining the transition as a delay. We have not determined the delay; is the delay always 5 seconds? Also, if the delay is always the same, would it not make more sense to have the transition regulated by a clock? If we clock our Finite-state Automata, we could then change the transition input to a loop that is only triggered depending on if the sensor does or does not see a car stopped on the street. So we would shift the diagram to figure 6, where we have extra states added to represent the consistent time delay, and we only have the transition values of a car or no car, so our set Q only contains the values of Car or No Car. Likewise, we would only have to list the states beneficial to the machine's functionality. So in our updated Finite-state Machine diagram, we could remove the error states like green and green, but we would need to add some new states to represent the automated delay due to the clocked nature of this automata. The following mathematical characteristics would describe the result.

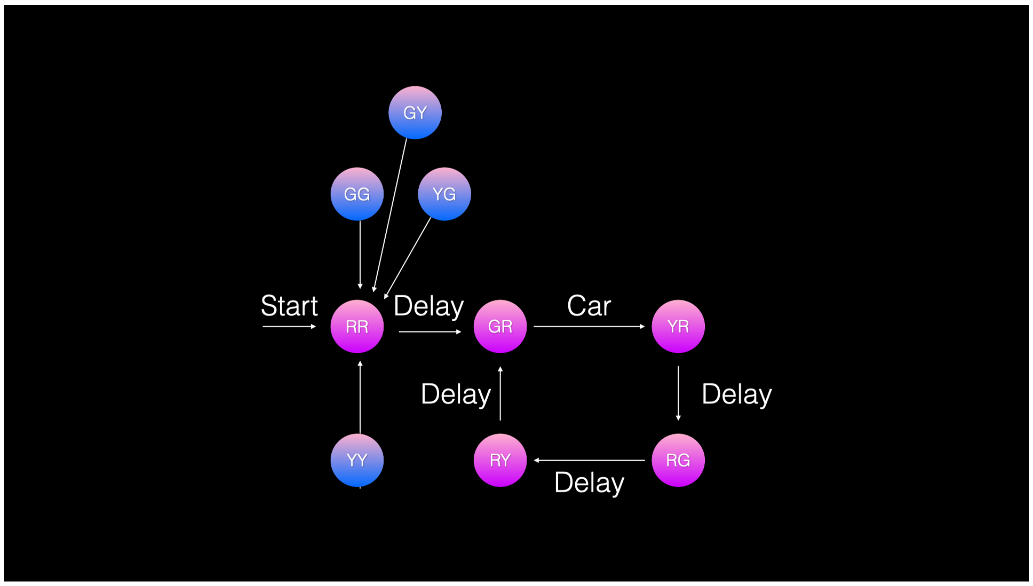


Figure 4: Traffic Light Finite-state Machine

1. Σ = {Car,No Car}
2. Q = {RR, RR1, GR, YR, YR1, RG, RG1, RY, RY1}
3. δ: $Q \times \Sigma \to Q$, where δ is defined as in the figure for this Finite-State Machine.
4. The starting state is $RR \in Q$
5. The set of accepting states is $A = \{GR\} \subseteq Q$

These characteristics would result in the updated diagram as in figure 5. We can then reference our transition table as demonstrated in table 1, better understanding how you would transition to the next state at the given state.

We can see from this table there is only one state that the machine can be in, whereas the next state is dependent on the input. This means that there will be only one row in our Karnaugh map, which defeats the point of the map to minimize the number of states and inputs of the machine.

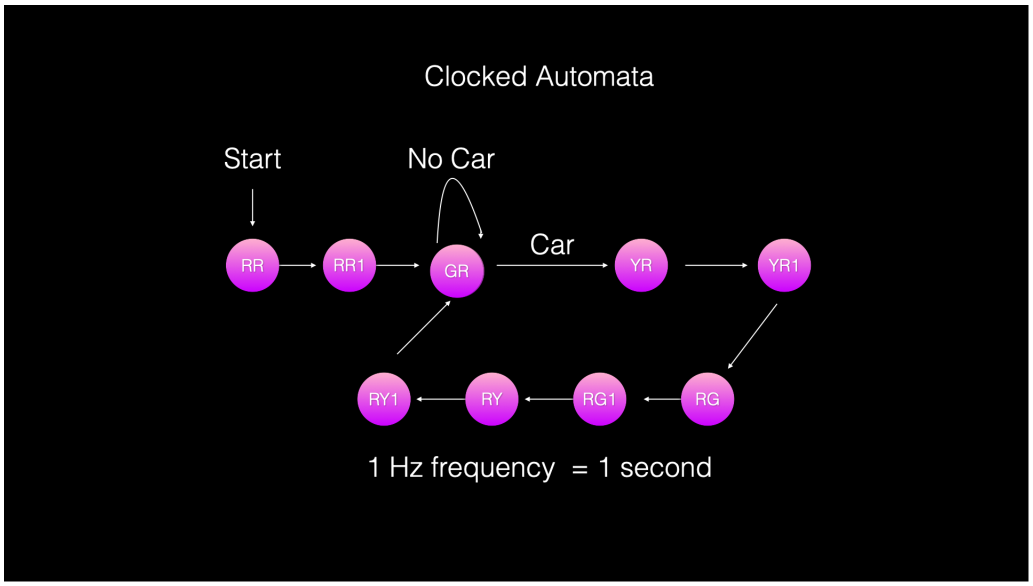


Figure 5: Clocked Traffic Light Finite-state Machine

Q	Σ	$\to$	Q
GR	No Car	$\to$	GR
GR	Car	$\to$	YR

Table 1: Transition Table for Traffic Light

So we will not write a Karnaugh map in this case because the transition table demonstrates that we are already at the minimum amount of components [2].

5 Circuit Case Example: D Flip-Flop

Let us consider a final example of an interesting application of a Finite-state Machine, a D Flip-Flop. Note that the input **Q** and the output **Q+** are in boldface to differentiate between the set and the elements.

As you may know, D Flip-Flops are important circuit pieces because they can keep and store

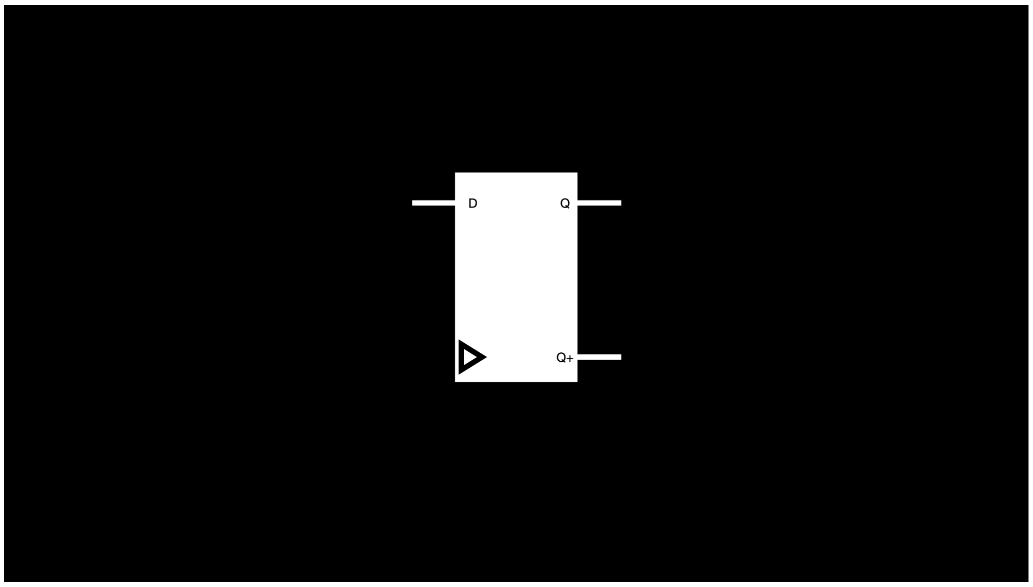


Figure 6: The Diagram of a D Flip-Flop circuit

a bit until the clock edge rises for access to remove and replace the last known value. Recall that a D Flip-Flop circuit looks like the diagram in figure 6, with a D and **Q** input, a clock input, and a **Q**+ output. We can think about how the combination of inputs can affect the output **Q**+ by building a transition table like the one below. We can expect that we will have four rows because we have two inputs, and each input will have two possible combinations of those inputs, i.e., 2^2 possible combinations. From this table, we can define our characteristic equation of the operation of this machine as $\mathbf{Q}+ = D$ (**Q**+ is dependent on D). This is helpful in our understanding of how to draw the Finite-state Machine diagram as we can define our transition values of our set Σ to be either D or Not D ($\overline{D}$). However, we still have the **Q** input, which contributes to the final output **Q**+ so whether or not **Q**, we can determine that **Q** or not **Q** ($\overline{\mathbf{Q}}$) would be our states instead of being a transition value. We now have deduced enough information to characterize this Finite-state Machine.

D	**Q**	**Q**+
Not D	*Not* **Q**	*Not* **Q**
Not D	**Q**	*Not* **Q**
D	*Not* **Q**	**Q**
D	**Q**	**Q**

Table 2: Transition Table for D Flip-Flop

1. Σ = {D,$\overline{D}$}
2. Q = {$\mathbf{Q}$, $\overline{\mathbf{Q}}$}
3. δ: $Q \times \Sigma \rightarrow Q$, where δ is defined as in the figure for this Finite-State Machine.
4. The start state is $\overline{\mathbf{Q}} \in Q$
5. The set of accepting states is $A = \{\mathbf{Q}, \overline{\mathbf{Q}}\} \subseteq Q$

We can see that our Finite-state Machine diagram will be straightforward and look like figure 7 below.

As we can see, as simple as the machine is, we can now create the Karnaugh Map to verify that we have a minimal amount of components in our machine. As seen below in figure 8, there are only two states we can switch between in our machine, so we have minimal components [1].

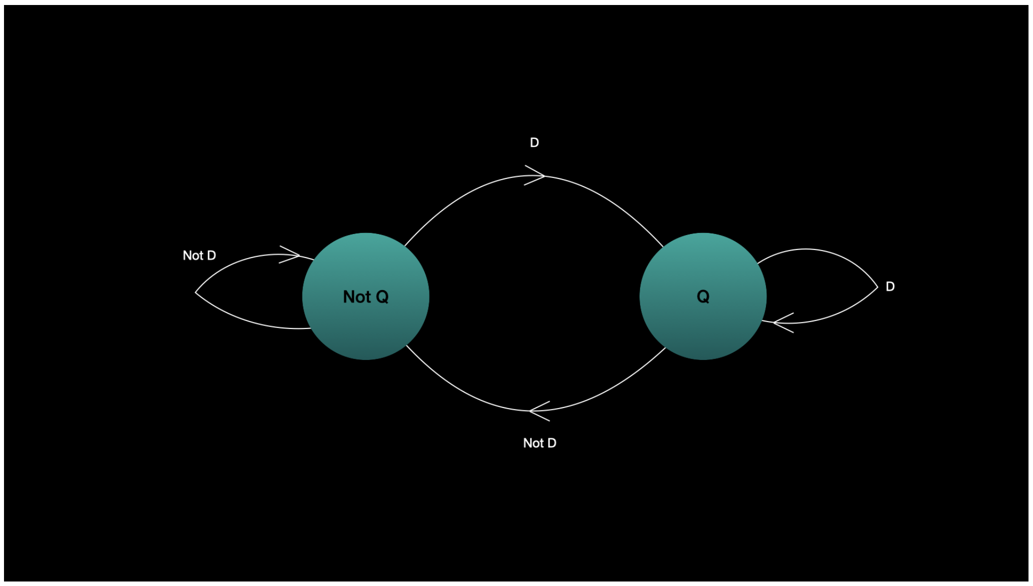
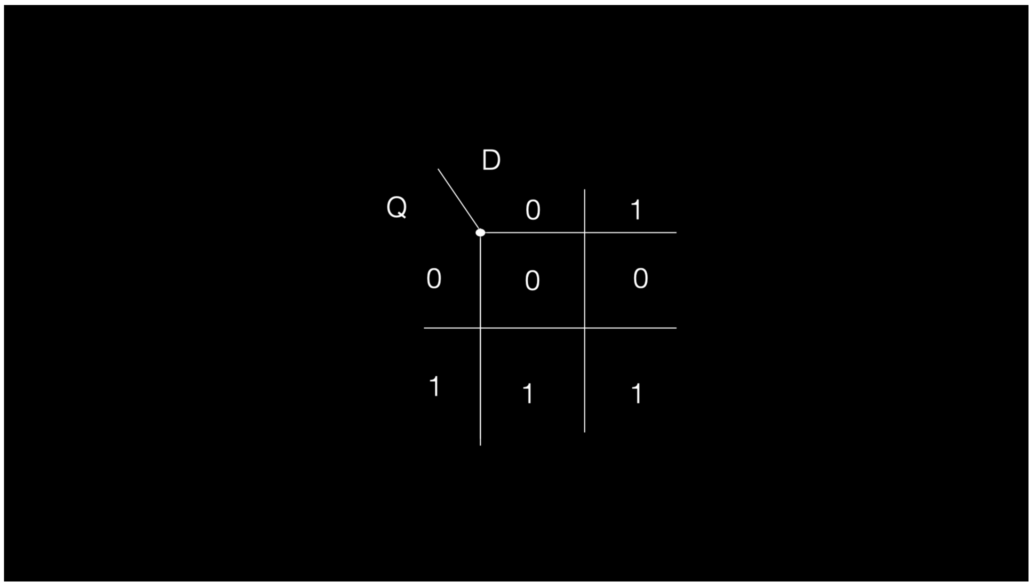


Figure 7: Diagram of D Flip-Flop Finite-state Machine



Q D	0	1
0	0	0
1	1	1

Figure 8: The Diagram of a D Flip-Flop circuit

6 Summary

It is clear that there are Finite-state Machines everywhere in our world, and it can be helpful to understand the best way to make our world work more efficiently. The intersection of mathematics, electrical engineering, and computer science demonstrates how interconnected our daily lives are with computers and how dependent, as a society, we are on them.

References

[1] Tertulien Ndjountche, *Digital Electronics Volume 3 Finite-State Machines*, Wiley, 2016.

[2] Peter Kogge, *The Zen of Exotic Computing*, Self Published, 2022, https://books.google.com/books?id=rPV_zwEACAAJ

[3] Jiacun Wang, *Handbook of Finite State Based Models and Applications*, Taylor and Francis, 2012, https://books.google.com/books?id=rPV_zwEACAAJ

[4] Jeff Erickson, *Algorithms* , Self Published, 2019, http://jeffe.cs.illinois.edu/teaching/algorithms/models/all-models.pdf

[5] Douglas Thain, *Ch. 3 Scanning*, Introduction to Compilers and Language Design, 2nd ed., Self Published, 2021, https://www3.nd.edu/~dthain/compilerbook/.

[6] Rania Hussein, *Lab 3 Boolean Algebra and K-maps*, University of Washington, 2022, https://rhlab.ece.uw.edu/wp-content/uploads/sites/35/2022/03/Lab-3_-Boolean-Algebra-and-K-Maps.pdf

[7] Mátyás Lancelot Bors, *"What Is a Finite-state Machine?"*, Medium, 10 March 2018, https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c.

[8] Francis E. Su, et al., *“Pigeonhole Principle.”*, Math Fun Facts, https://math.hmc.edu/funfacts/pigeonhole-principle/.

[9] Rafael Pass, Wei-Lung Dustin Tseng, *“A Course in Discrete Structures.”*, Cornell University, 2021, https://www.cs.cornell.edu/~rafael/discmath.pdf.

[10] The Editors of Encyclopaedia Britannica, "Stephen Cole Kleene", Encyclopedia Britannica, 21 Jan. 2022, https://www.britannica.com/biography/Stephen-Cole-Kleene.

[11] Amal Dar Aziz, et al., "Automata Theory", *Basics of Automata Theory*, Stanford Univerity, 2004, https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html.

[12] ECStudioSystems, ‘’D Flip-Flop‘‘, *ECStudioSystems.com*, ECStudioSystems 2021 D Flip-Flop, https://ecstudiosystems.com/discover/textbooks/basic-electronics/flip-flops/d-flip-flop/.