Basic Finite State Machines

With Examples in Logisim and Verilog

By: Andrew Tuline

Date: June 4, 2013

This is a work in Progress!

Introduction

Having recently rekindled my interest in electronics, I decided to re-learn various aspects of digital logic. This document provides some examples of the analysis and design of a few simple Finite State Machines.

What Is Covered

- Moore machines
- Mealy machines
- Logisim (as in free software) based circuit designs
- Verilog based circuit designs using Altera's Quartus II (the free version)
- Verilog based testbench using Altera/Mentor's ModelSim (also free)

What Is Not Covered

This document assumes you are already familiar with:

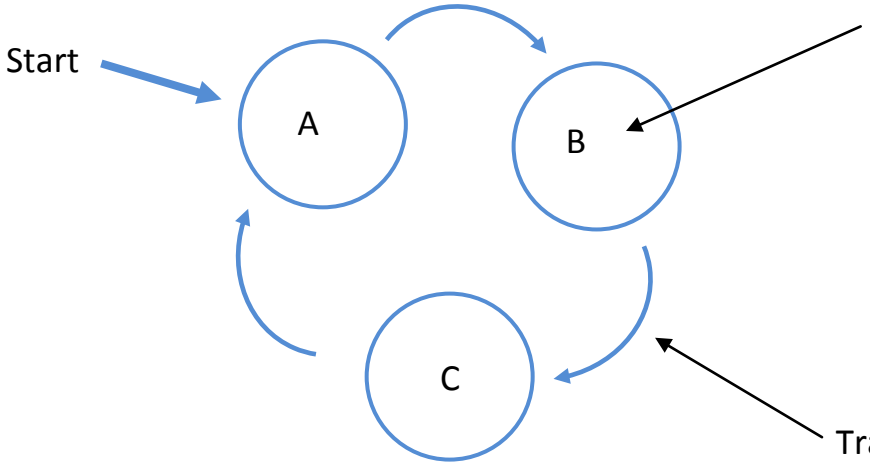
- Boolean logic
- Number systems
- Flip-flop basics
- Truth tables
- Karnaugh maps
- Combinational logic
- Hazards (timing glitches)
- Ripple vs synchronous counters

In addition, this document does not cover:

- Verification of the design
- Timing analysis of the design

A Simple Finite State Machine

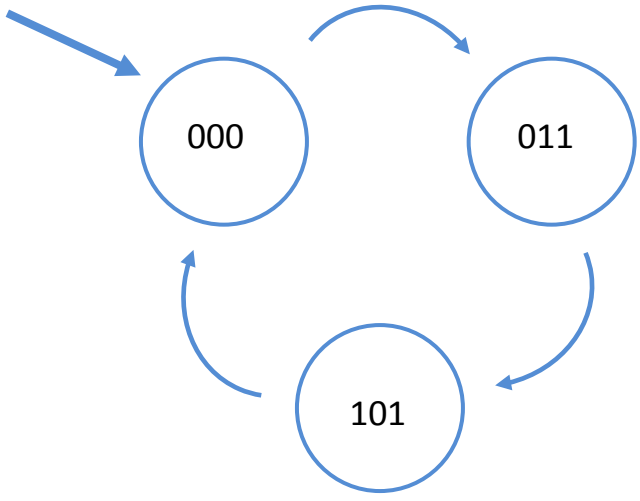
Whether it be a counter, a sequence recognizer, a vending machine or an elevator, through the use of combinational and sequential logic, we can store information about a system in the form of a Finite State Machine. Here's a very simple example of a Finite State Machine that changes states without any additional inputs or outputs. It's a counter:



State Name	Binary Representation
A	00
B	01
C	10

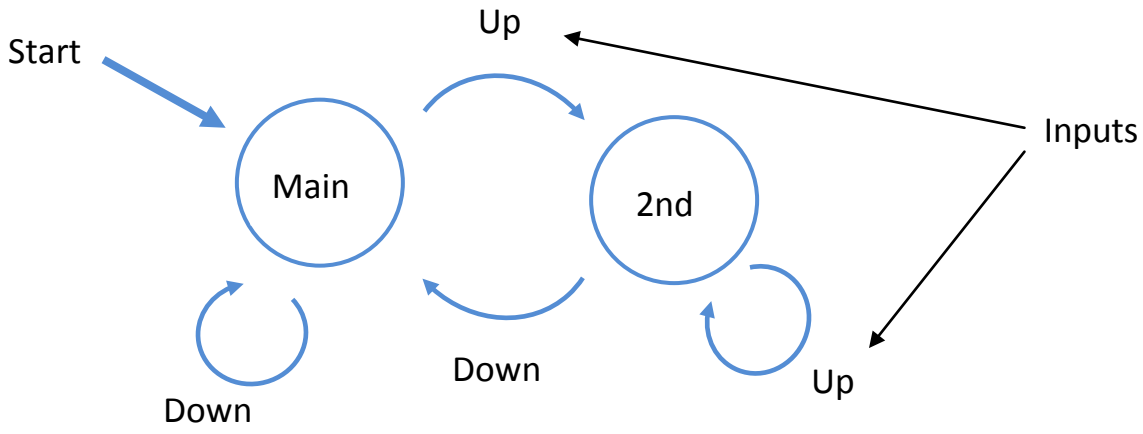
This simple Finite State Machine, or 'FSM' has 3 states, A, B and C. This will automatically transition between each state with a clock signal. These states can be represented in binary with 2 bits, supported by 2 flip-flops which would be used to store the state information. The blue arrow points to the 'starting' state.

These 3 states could also contain more than 2 bits. For example:



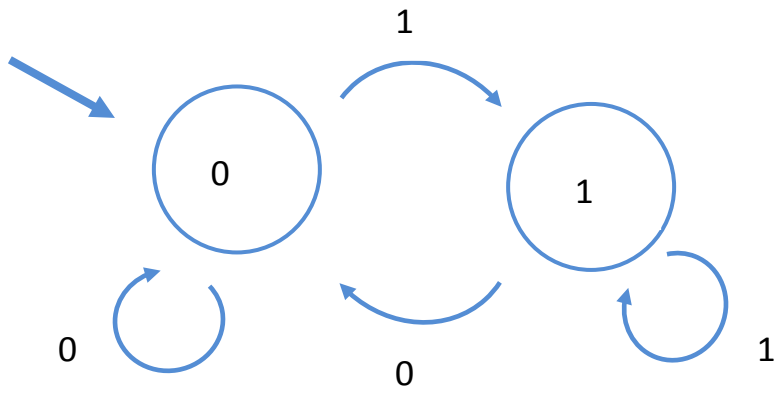
Although, there are only 3 states (thus 2 bits), there happen to be 3 bits of stored information, therefore 3 flip-flops would be used for this FSM.

In addition, you can use inputs to move from one state to the next. Let's say we have an elevator with 2 buttons, 'Up' and 'Down'. This could be represented as follows:



If you're on the main floor and press the 'Up' button, you go up. If you press it again, nothing happens. Of course, the 'down' button takes you down.

Let's create binary representation of the inputs and states as follows:



State Name	Binary Representation
Main	0
2nd	1

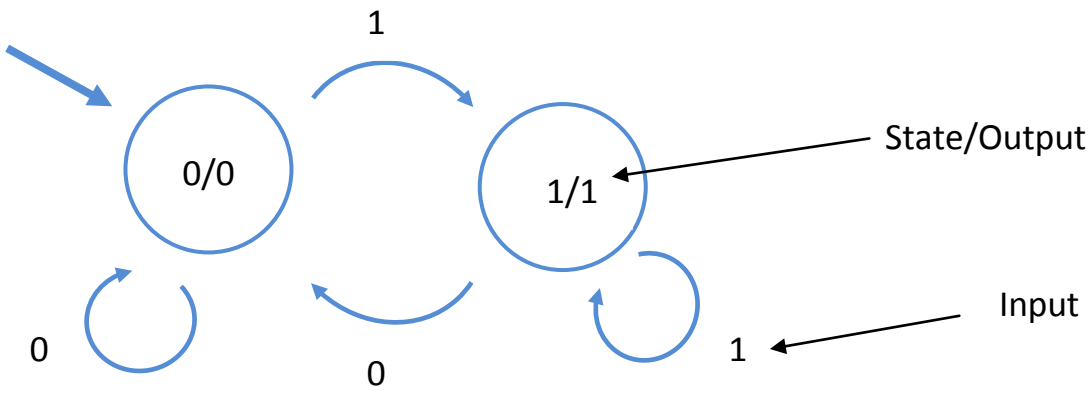
Button/ Input	Binary Representation
Down	0
Up	1

Mealy and Moore Machines

Mealy and Moore machines are used to represent our elevator as finite state machines. These provide:

- States
- State transitions
- Inputs
- Outputs

In the previous examples, we would have used the state value as our circuit 'output'. Shown below is a Moore machine where the output values are determined by its current state:

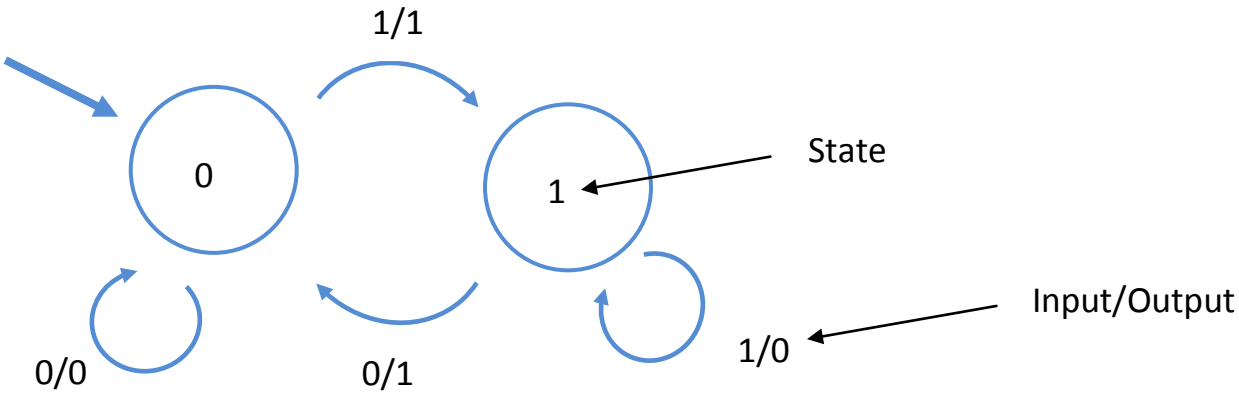


In this case, it's just a coincidence that the output and state values match.

What if you wanted to:

- Push the ‘1’ button to go up, and the ‘0’ button to go down, and
- Output a ‘1’ every time you change floors and a ‘0’ when you don’t.

This can be shown by a Mealy machine, which displays the outputs on the state transition instead of the state itself, as shown below:



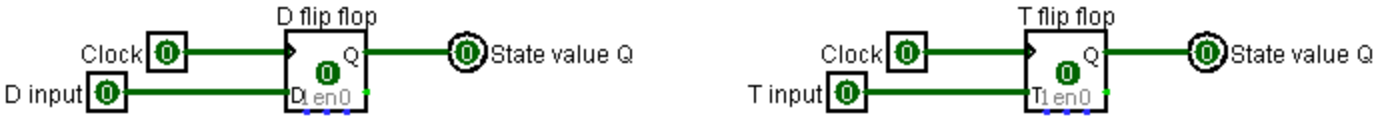
As you can see, it’s easy to represent this change with a Mealy machine, but you would require additional states to represent this with a Moore machine. Therefore, you might want to use a Moore machine when outputs are just associated with the state and a Mealy machine when the outputs are associated with an input as well as the state.

Flip-flops We'll Use

In order to save the state information, we'll use flip-flops. There are several available from which to choose, such as RS, D, T, JK, and several options for each, such as enable, preset, clear or even latches. For this article, we'll focus on basic 'D' and 'T' flip-flops. In addition, we'll be simulating our resultant circuits with Logisim and later with Verilog and ModelSim. See the references at the end of this document for information on downloading this free software.

A 'D' flip-flop is usually used as a register, where the next state takes on the value of the current input.

A 'T' flip-flop is usually used as a counter, where the next state toggles if the current input is a '1'.



We can also configure a JK flip-flop as both a 'T' and 'D' as follows (with Logisim):

We'll also be using a Next State Table as shown below in order to determine the logic required in order to create our FSM's.

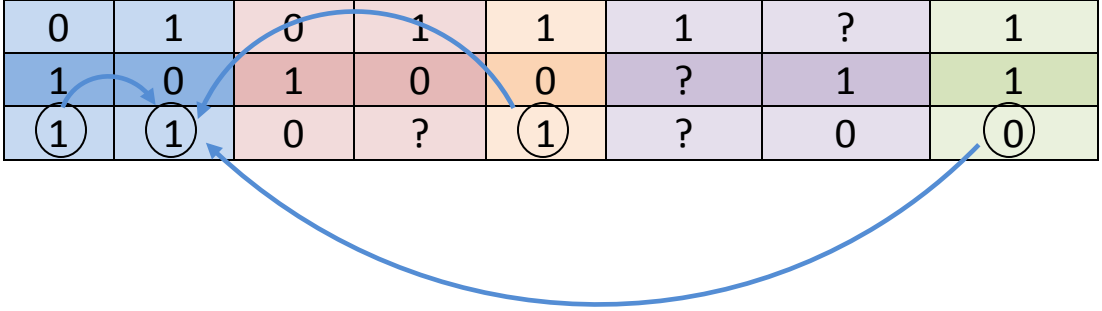
Q	Q+	R	S	D	J	K	T
0	0	?	0	0	0	?	0
0	1	0	1	1	1	?	1
1	0	1	0	0	?	1	1
1	1	0	?	1	?	0	0

Note: The Survivalcraft game for the Android contains SR flip-flops, which are actually JK. As a result, you can convert them to 'T' or 'D' and use the examples in this document in the game.

Next State Transition Table

For each flip-flop, we'll need to develop a good understanding of the current state values (or Q) and the inputs required to generate the next state value (or Q+).

Q	Q+	R	S	D	J	K	T
0	0	?	0	0	0	?	0
0	1	0	1	1	1	?	1
1	0	1	0	0	?	1	1
1	1	0	?	1	?	0	0

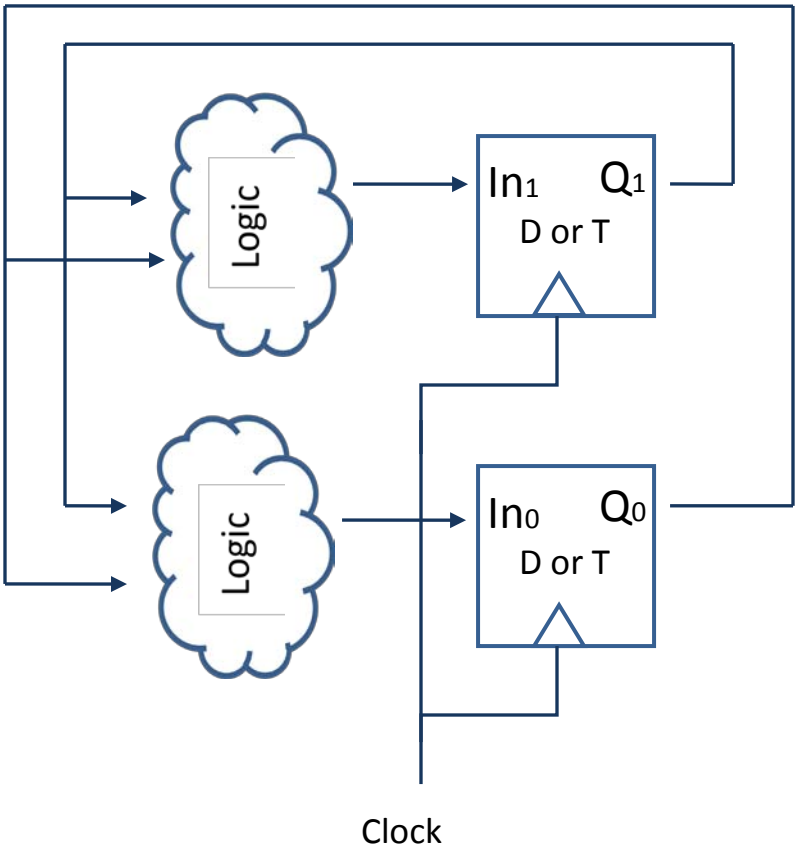


In the above table, For the current state Q=1 to change in the next state to Q+=1 during a clock cycle, a 'D' flip-flop would require an input value of 1, whereas a 'T' flip-flop would require an input value of 0.

From there, we'll build a table of values required in order to accomplish the outputs of our FSM. After that, we build Karnaugh mapping tables in order to determine the logic.

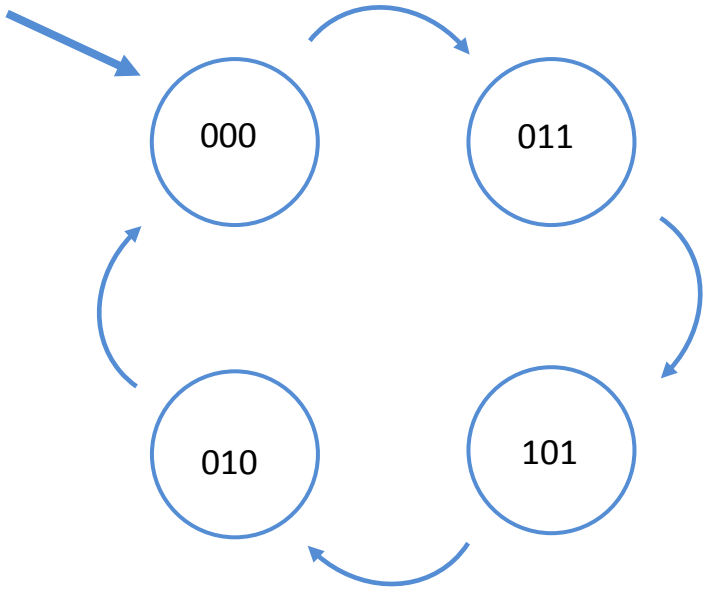
What Our First Circuit Will Look Like

We'll use combinational logic to derive the 'D' or 'T' input value from the current state values. In addition, more advanced circuits will include input and separate output values in the logic.



Example 1: Our Counter

Description: On each clock cycle, the counter will change to the next state.



Let's represent the counter states with 3 bits called Q2, Q1 and Q0. The 'next' states will be referred to as Q2+, Q1+ and Q0+. We'll need to create a current and next state table with these Q values as follows:

Q2	Q1	Q0
0	0	0
0	1	1
1	0	1
0	1	0

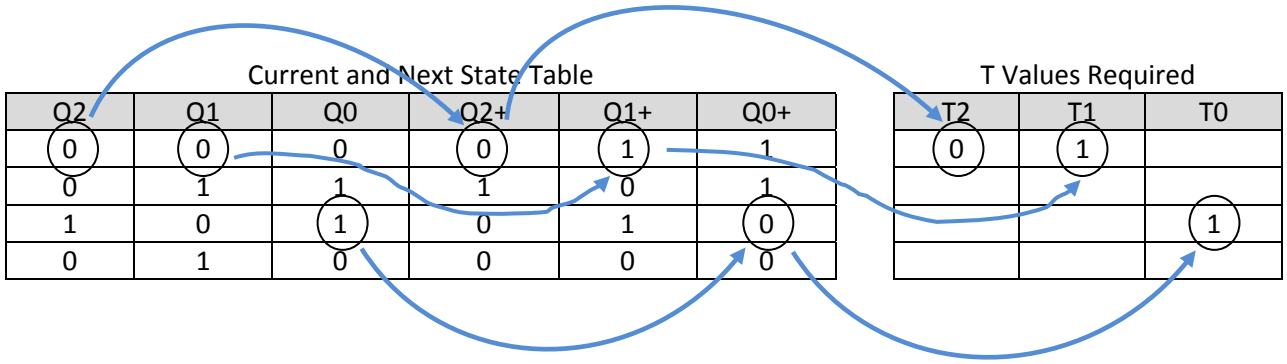
Q2+	Q1+	Q0+
0	1	1
1	0	1
0	1	0
0	0	0

Next, we will need to determine which type of flip-flop we're going to use in our circuit (we'll try both 'T' and D). We'll then need to create a table that shows the inputs required in order to progress to the next state.

T Flip-flop Version

Since a 'T' flip-flop is generally used as a counter, let's try that example first. Here's a 'T' flip-flop based transition table:

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0



In this example

- The topmost Q2 is a 0, and Q2+ is a 0, so the 'T' value required to get there will be a 0.
- The topmost Q1 is a 0, and the Q1+ is a 1, so the 'T' value required will be a 1.
- Finally, the third Q0 entry is a 1, while the Q0+ entry is a 0, so the 'T' value required will be a 1.

Let's fill in the State table, and include the 'T' value inputs for each.

Current and Next State Table

Q2	Q1	Q0	Q2+	Q1+	Q0+
0	0	0	0	1	1
0	1	1	1	0	1
1	0	1	0	1	0
0	1	0	0	0	0

T Values Required

T2	T1	T0
0	1	1
1	1	0
1	1	1
0	1	0

Next, we'll need to generate Karnaugh maps (generation tables) with the Q2, Q1, Q0 outputs for each of T2, T1 and T0.

We'll put an 'x' (don't care) in each location that isn't listed above.

T2 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	0	0	x	x
1	x	1	x	1

T2 = Q0

T1 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	1	1	x	x
1	x	1	x	1

T1 = 1

T0 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	1	0	x	x
1	x	0	x	1

T0 = 'Q1 (as in NOT Q1)

Let's create a circuit based on this design, with 'T' flip-flops.

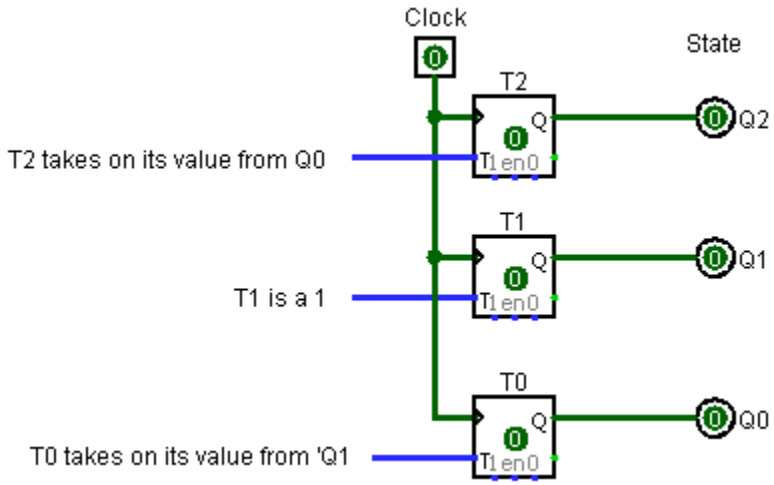
According to the generation tables:

- T2 takes on its value from Q0
- T1 is a 1
- T0 takes on its value from 'Q1

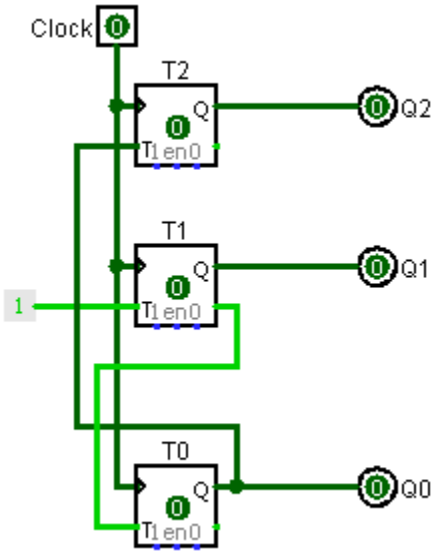
All we need to do now is to toggle the clock in Logisim and watch the Q values go from:

- 000
- 011
- 101
- 010

Here's the basic circuit from our tables:



Let's get those inputs hooked up from the appropriate pins as follows:



As an exercise in Logisim, set the Q outputs to all the different possible states and toggle the clock. Although this circuit is quite simple, this counter may not be able to 'reset' itself if it starts in the wrong state. As a result, we may have to go in and tighten the logic up by excluding some of the 'don't care' states in our Karnaugh map in order to make it more reliable. Good luck!

D Flip-flop Version

Now, let's try the circuit with some 'D' flip-flops instead.

Current and Next State Table

Q2	Q1	Q0	Q2+	Q1+	Q0+
0	0	0	0	1	1
0	1	1	1	0	1
1	0	1	0	1	0
0	1	0	0	0	0

D Values Required

D2	D1	D0
0	1	1
1	0	1
0	1	0
0	0	0

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Again, we'll need to generate Karnaugh maps with the Q2, Q1, Q0 states for each of D2, D1 and D0.

We'll put an 'x' (don't care) in each location that isn't listed above.

D2 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	0	0	x	x
1	x	1	x	0

D2 = Q1 & Q0

D1 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	1	0	x	x
1	x	0	x	1

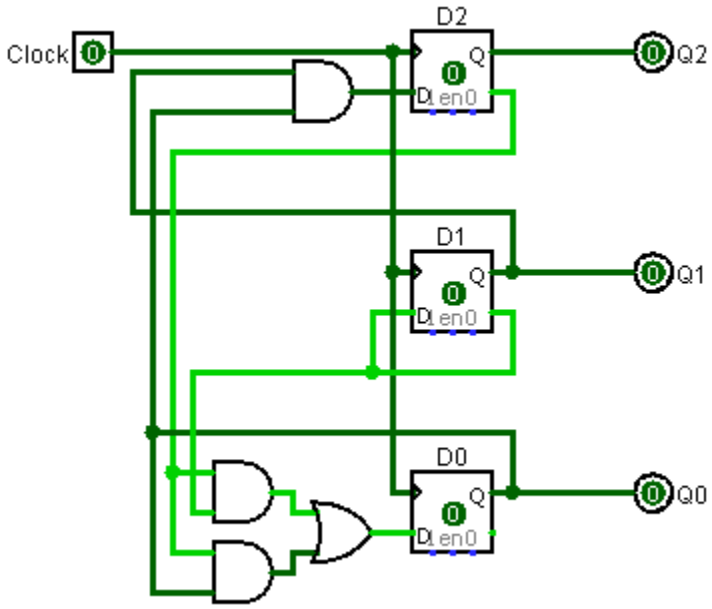
D1 = 'Q1

D0 Generation Table (from Q2/Q1/Q0 values)

	Q2/Q1			
Q0	00	01	11	10
0	1	0	x	x
1	x	1	x	0

D0 = 'Q2 & 'Q1 | 'Q2 & Q0

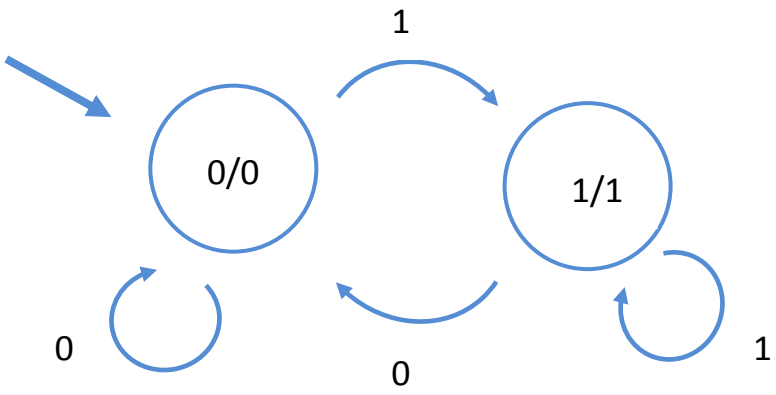
So, what does our resultant 'D' flip-flop logic look like?



Well, that's not very pretty. Again, change the clock state to watch it count up through the predicted states. Unlike our 'T' based counter, this one should correct itself pretty quickly if put into a random state, but it's not nearly as simple a design as the 'T' version.

Example 2: The Elevator

Our elevator from the introduction has fewer states than the counter, but does have an input to change states as well as an output value. This incarnation is a Moore machine, where the output is just a function of the state.



Analysis

First off, we'll need to come up with our current/next state table, which will include the Output. We're also going to design this with 'D' and 'T' flip-flops as well as Verilog.

Q0	Input	Q0+	Output
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

⇨

D0
0
1
0
1

T0
0
1
1
0

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

From the current/next state table, along with ‘D’ and ‘T’ state transition tables, we can determine the required values of D0 and T0.

D0 Table (from Q0/In)

	Q0	
Input	0	1
0	0	0
1	1	1

D0 = In

T0 Table (from Q0/In)

	Q0	
Input	0	1
0	0	1
1	1	0

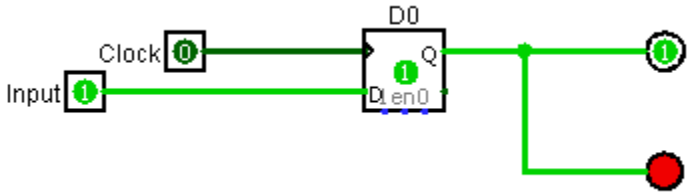
T0 =(In & 'Q0) | ('In & Q0)
or
T0 = In ^ Q0 (as in xor)

Output Table (from Q0/In)

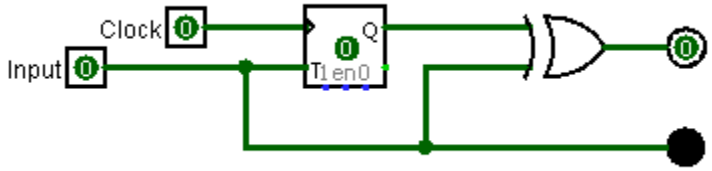
	Q0	
Input	0	1
0	0	0
1	1	1

Output = In

Here’s our ‘D’ flip-flop version done with Logisim.



Here’s our ‘T’ flip-flop version. It more complex if we didn’t see that an xor gate could be used.



Summary

It’s interesting to see how the ‘D’ flip-flop scenarios differ from the ‘T’ versions. Let’s try designing the elevator using Quartus II and Verilog next.

The Verilog Method

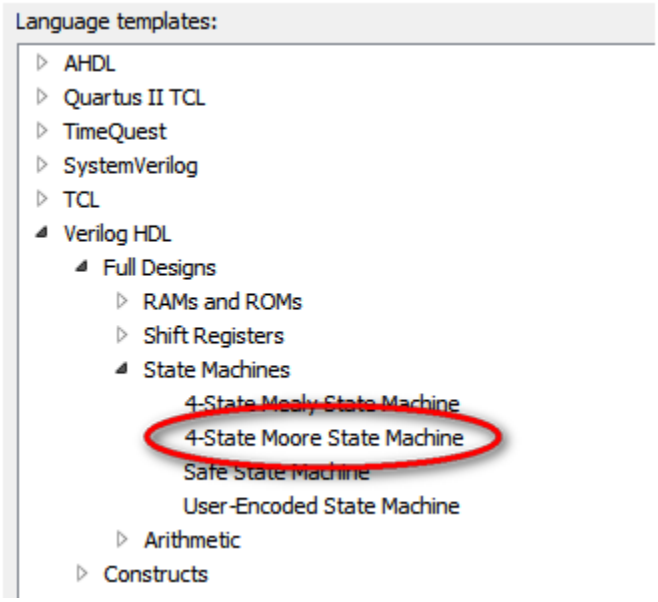
This section assumes you are already familiar with using Verilog for both basic combinational and sequential designs. See my Verilog tutorial if you aren't.

In order to design our elevator using traditional logic, we had to perform a significant amount of analysis, develop truth tables and Karnaugh maps and then convert it all to sequential and combinational logic. Developing an FSM with Verilog uses a completely different approach and is actually significantly easier once you understand the template that's used. In fact, you can go directly from the Mealy/Moore diagram to your Verilog code.

Altera's Quartus II software provides a template to get you started developing a FSM on your own. To do so:

1) Open or create a project in Quartus II.
2) Open or create a new Verilog HDL file (and save it).
3) Click on the 'scroll' icon as shown:

4) Insert a 4 state Moore Machine as shown :



The Quartus template provides a working 4 state Moore FSM (which I won't duplicate). Here is the core functionality of this template in Verilog:

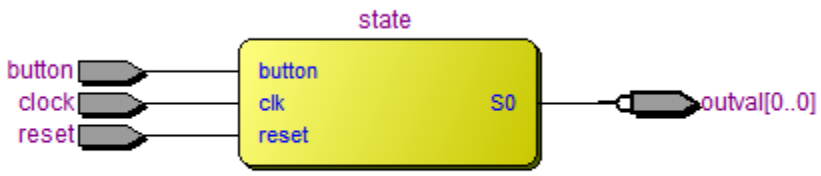
```
module moore_state_machine  (
	// Declare inputs and outputs
);

	// Define the state register
	//

	// Define the states
	//

	// Sequential section to provide output for each state
	//

	// Sequential section to determine the next state
	//

endmodule
```

Here's our Moore machine elevator in Verilog (called elevator.v):

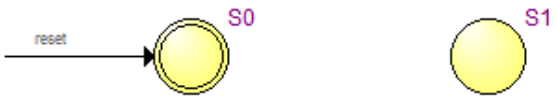
```
module elevator  (
        input    clk, in, reset,
        output reg [0:0] out
);

        reg      [0:0] state;                          // Define the state register  (1 bit wide)

        parameter S0 = 0, S1 = 1;                      // Define the states

        always @ (state) begin                         // Output depends only on the state
                case (state)
                        S0:
                                out = 1'b0;            // A single bit, which is '0'
                        S1:
                                out = 1'b1;            // A single bit, which is '1'
                        default:
                                out = 1'b0;
                endcase
        end

        always @ (posedge clk or posedge reset) begin  // Determine the next state
                if (reset)
                        state <= S0;                   // Support for reset
                else
                        case (state)
                                S0:
                                        if (in)
                                                state <= S1;
                                        else
                                                state <= S0;
                                S1:
                                        if (in)
                                                state <= S1;
                                        else
                                                state <= S0;
                        endcase
        end

endmodule
```

It might look a bit complicated at first, however this is a template that can easily be used for more complicated Moore machines, and you DON'T have to go through all that analysis. You can actually develop the Verilog logic directly by looking at each state and entering the next state and outputs from your Moore machine.

Here's what it looks like with the Netlist RTL viewer:



When you click on the yellow box, you can then see it in more detail in the State Machine viewer. Disappointingly, it does NOT represent our elevator's functionality as expected:

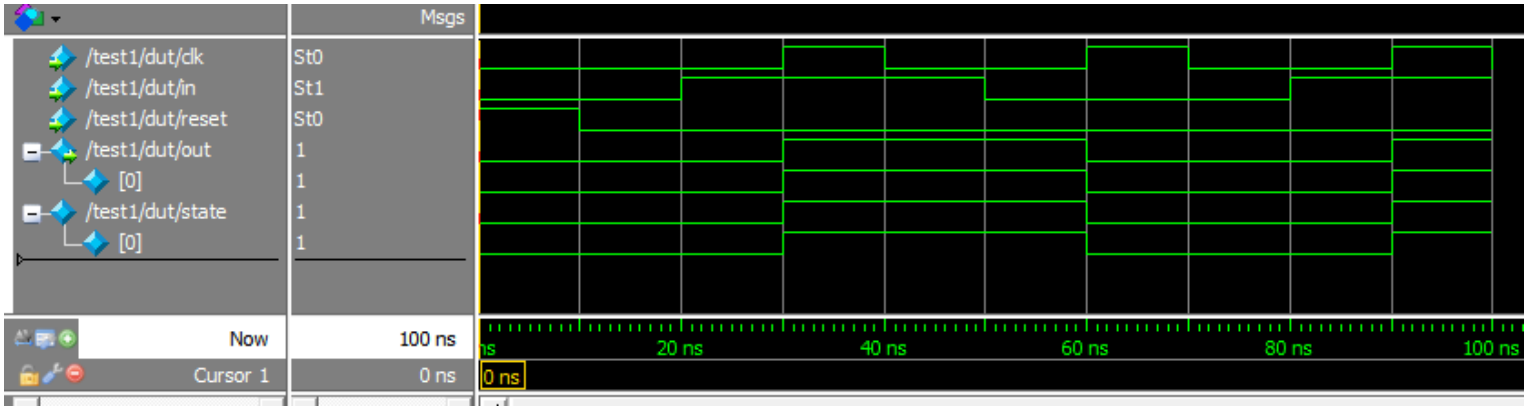


Let's try it out in ModelSim to see if/how it works. Before doing so, we'll need to create a testbench in order to test this design. We'll create and apply the clock along with input signals at appropriate times and view the resultant waveforms.

Here's our test bench (called test1.v):

```
// Elevator FSM test bench
//

`timescale 1ns/1ns

module test1 ();

        reg clk, in, reset;

        elevator dut (clk, in, reset, out);

        initial begin                                   // Here's the results from our waveform below
                clk=0; in=0; reset=1;   #10;            // 0ns - Clear everything and assert reset
                                                        //       everything at 0ns. State, output = 0.
                reset = 0;              #10;            // 10ns - Clear reset .
                in = 1;                 #10;            // 20ns - Set input.
                clk = 1;                #10;            // 30ns - Assert clock. State and output change
                                                        //        to 1.
                clk = 0;                #10;            // 40ns - De-assert clock.
                in = 0;                 #10;            // 50ns - De-assert input. State and output, no
                                                        //        change.
                clk = 1;                #10;            // 60ns - Assert clock. State and output change
                                                        //        to 0.
                clk = 0;                #10;            // 70ns - De-assert clock.
                in = 1;                 #10;            // 80ns - Assert input.
                clk = 1;                #10;            // 90ns - Assert clock. State and output change
                                                        //        to 1. Yeah!!!!
                clk = 0;                #10;            // And so on . .
                in = 0;                 #10;
                clk = 1;                #10;
                clk = 0;                #10;
        end

endmodule
```

Here's the ModelSim waveform:

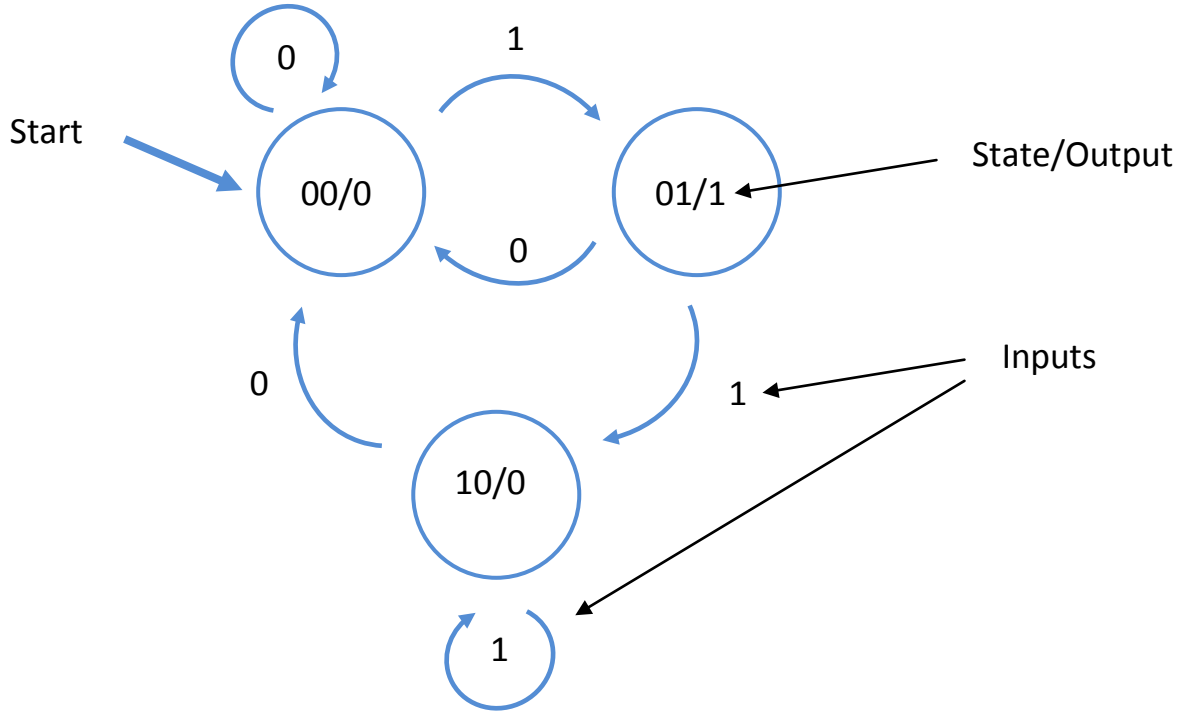


Summary

1) Although the Quartus State Machine Viewer doesn't show it, our elevator FSM works as designed in ModelSim.
2) Once understood, the Moore machine Verilog template is much easier to work with than going through the analysis performed for traditional logic.
3) We are avoiding risky 'x' states with this design.

Example 3: A ‘Gloppita’ Machine

Let’s say, we’ve been provided with the Moore machine outlined below. We don’t know what it does, but it doesn’t look too difficult to analyze and turn it into hardware.



We’ll develop circuits for both ‘D’ as well as 'T’ flip-flop circuits again.

First off, we’ll need to come up with our current/next state table, which will include the Output as well as the ‘D’ and ‘T’ inputs.

Current and Next State Table (along with the Output)

Q1	Q0	Input	Q1+	Q0+	Output
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	x	x	x
1	1	1	x	x	x

D values required

D1	D0
0	0
0	1
0	0
1	0
0	0
1	0
x	x
x	x

T values required

T1	T0
0	0
0	1
0	1
1	1
1	0
0	0
x	x
x	x

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

From the current/next state table, along with ‘D’ and ‘T’ state transition tables, we can determine the required values of D1, D0, T1 and T0.

‘D’ Flip-flop Version

Let’s analyze the ‘D’ flip-flop version first. We’ll need to determine the logic required to generate the outputs for D1, D0 and the Output as follows:

D1 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	0	x	0
1	0	1	x	1

D1 = (Q0 & In) | (Q1 & In) ➔ (Q0 | Q1) & In

D0 Generation Table (from Q1/Q0/Input values)

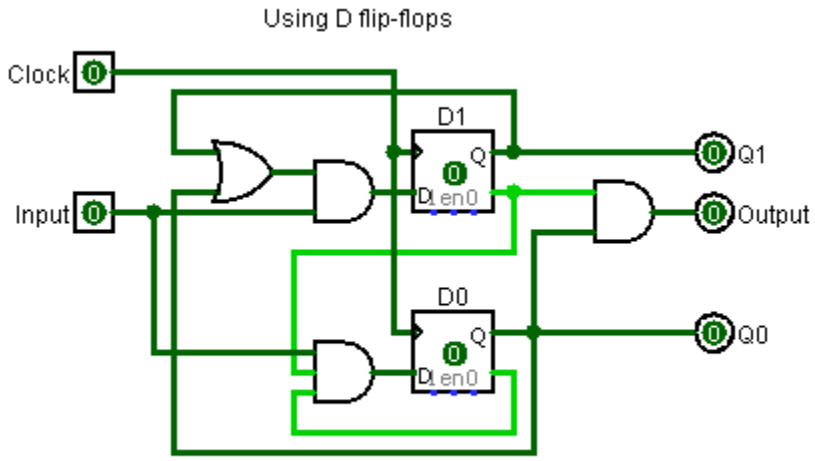
	Q1/Q0			
Input	00	01	11	10
0	0	0	x	0
1	1	0	x	0

D0 = 'Q1 & 'Q0 & In

Output Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	1	x	0
1	0	1	x	0

Output = Q0 (or even, 'Q1 & Q0)



I tried both the Output = Q0 as well as the Output = ('Q1 & Q0) versions, and both work nicely.

'T' flip-flop version.

T1 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	0	x	1
1	0	1	x	0

T1 = (In & Q0) | ('In & Q1)

T0 Generation Table (from Q1/Q0/Input values)

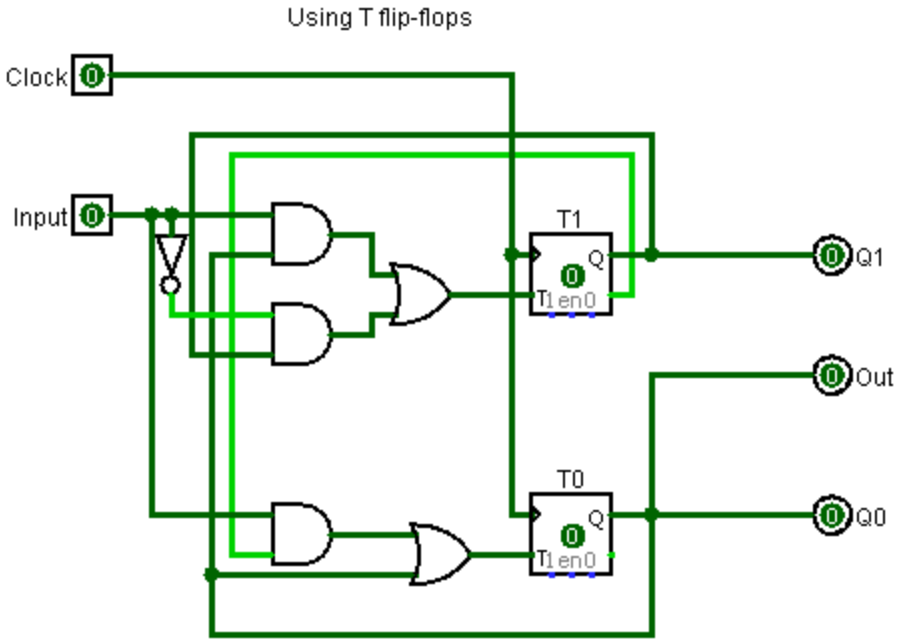
	Q1/Q0			
Input	00	01	11	10
0	0	1	x	0
1	1	1	x	0

T0 = (In & 'Q1) | Q0

Output Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	1	x	0
1	0	1	x	0

Output = Q0



Again, set the input value prior to toggling the clock for each state.

Verilog version

```
module gloppita  (
        input    clk, in, reset,
        output reg [0:0] out
);

        reg      [1:0] state;                    // Define the state register

        parameter S0 = 0, S1 = 1, S2 = 2;        // Define the states

        always @ (state) begin                   // Output depends only on the state
                case (state)
                        S0:
                                out = 2'b00;
                        S1:
                                out = 2'b01;
                        S2:
                                out = 2'b10;
                        default:
                                out = 2'b00;
                endcase
        end

        always @ (posedge clk or posedge reset) begin   // Determine the next state
                if (reset)
                        state <= S0;
                else
                        case (state)
                                S0:
                                        if (in)
                                                state <= S1;
                                        else
                                                state <= S0;
                                S1:
                                        if (in)
                                                state <= S2;
                                        else
                                                state <= S0;
                                S2:
                                        if(in)
                                                state <= S2;
                                        else
                                                state <= S0;
                        endcase
        end
endmodule
```

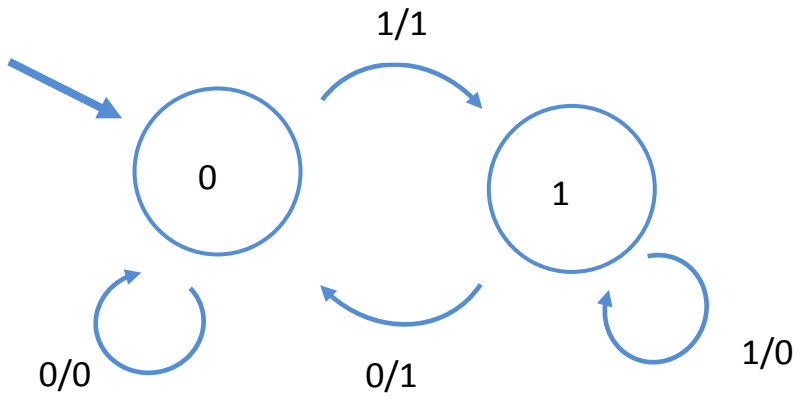
Summary

- This design was a few small edits different than our elevator design. This included widening the state register, increasing the number of states by 1 and making small modifications to the output as well as the next state logic.
- Again, the State Machine Viewer didn't represent this very well.
- We didn't have to go through the full analysis in order to develop this.

Example 4: A Mealy Elevator

Our original elevator was a Moore machine, where the outputs were tied to the state. In this case, the outputs are a function of the state AND the input, thus we have a Mealy machine.

- Push the '1' button to go up, and the '0' button to go down.
- Output a '1' every time you change state, and a '0' when you don't.



Analysis

Please have a go at this yourself.

Q0	Input	Q0+	Output
0	0	0	
0	1	1	
1	0	0	
1	1	1	

⇨

D0	T0

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

From the current/next state table, along with ‘D’ and ‘T’ state transition tables, we can determine the required values of D0 and T0.

D0 Table (from Q0/Input)

Q0		
Input	0	1
0		
1		

D0 =

T0 Table (from Q0/Input)

Q0		
Input	0	1
0		
1		

T0 =

Output Table (from Q0/Input)

Q0		
Input	0	1
0		
1		

Output =

Here’s our ‘D’ flip-flop version done with Logisim.

Here’s our ‘T’ flip-flop version done with Logisim.

Summary

Congratulations on designing your first Mealy machine!

Verilog version

This looks very similar to the Moore version, but allows for additional logic with each case comparison.

If you look at the 'always' statements for this machine, you'll see that:

1) The next state is determined synchronously with the clock
2) More importantly, the output does not wait for the clock, but asynchronously happens immediately upon being set or reset.
3) You can easily change those 'always' conditions.

```
module mealy_elevator (
	input	clk, in, reset,
	output	reg [0:0] out
);

	reg	[0:0]state;				// Declare state register

	parameter S0 = 0, S1 = 1;			// Declare states

	// Determine the next state synchronously, based on the current state and the input
	always @ (posedge clk or posedge reset) begin
		if (reset)
			state <= S0;
			out <= 1'b0;
		else
			case (state)
				S0:
					if (in) begin
						state <= S1;
					end
					else begin
						state <= S0;
					end
				S1:
					if (in) begin
						state <= S1;
					end
					else begin
						state <= S0;
					end
			endcase
	end
```

```
// Determine the output based only on the current state and the input (do not wait for a clock edge).
	always @ (state or in)
	begin
			case (state)
				S0:
					if (in) begin
						out = 1'b1;
					end
					else begin
						out = 1'b0;
					end
				S1:
					if (in) begin
						out = 1'b0;
					end
					else begin
						out = 1'b1;
					end
			endcase
	end

endmodule
```

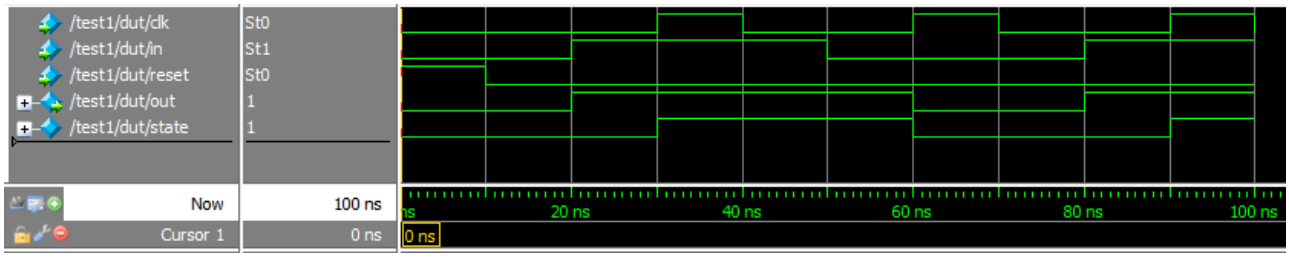
Summary

Yet again, the State Machine Viewer doesn't show our design correctly, so let's create a testbench in order to ensure it works as expected. We can use the same test bench that we'd designed for the Moore elevator.

Here's the testbench for our Mealy elevator with updated comments.

```
// Elevator FSM test bench
//

`timescale 1ns/1ns

module test1 ();

        reg clk, in, reset;

        elevator dut (clk, in, reset, out);

        initial begin                                   // Here's the results from our waveform below
                clk=0; in=0; reset=1;     #10;          // 0ns - Clear everything and assert reset
                                                        // everything at 0ns. State, output = 0.
                reset = 0;                #10;          // 10ns - Clear reset .
                in = 1;                   #10;          // 20ns - Set input, and output asynchronously
                                                        //        goes to 1.
                clk = 1;                  #10;          // 30ns - Assert clock. State changes to 1. Yeah!!
                clk = 0;                  #10;          // 40ns - De-assert clock.
                in = 0;                   #10;          // 50ns - De-assert input. Output goes to 0,
                                                        //        while state has no change.
                clk = 1;                  #10;          // 60ns - Assert clock. State and changes to 0.
                clk = 0;                  #10;          // 70ns - De-assert clock.
                in = 1;                   #10;          // 80ns - Assert input and output changes to 1.
                clk = 1;                  #10;          // 90ns - Assert clock. State changes to 1.
                clk = 0;                  #10;          // And so on . .
                in = 0;                   #10;
                clk = 1;                  #10;
                clk = 0;                  #10;
        end

endmodule
```

Here's the resultant waveform:



Summary

- Note that the output changes immediately upon the change of input, and not with the clock.
- Our Mealy machine works as designed.
- It’s still easy.

Example 4: A Sequence Recognizer

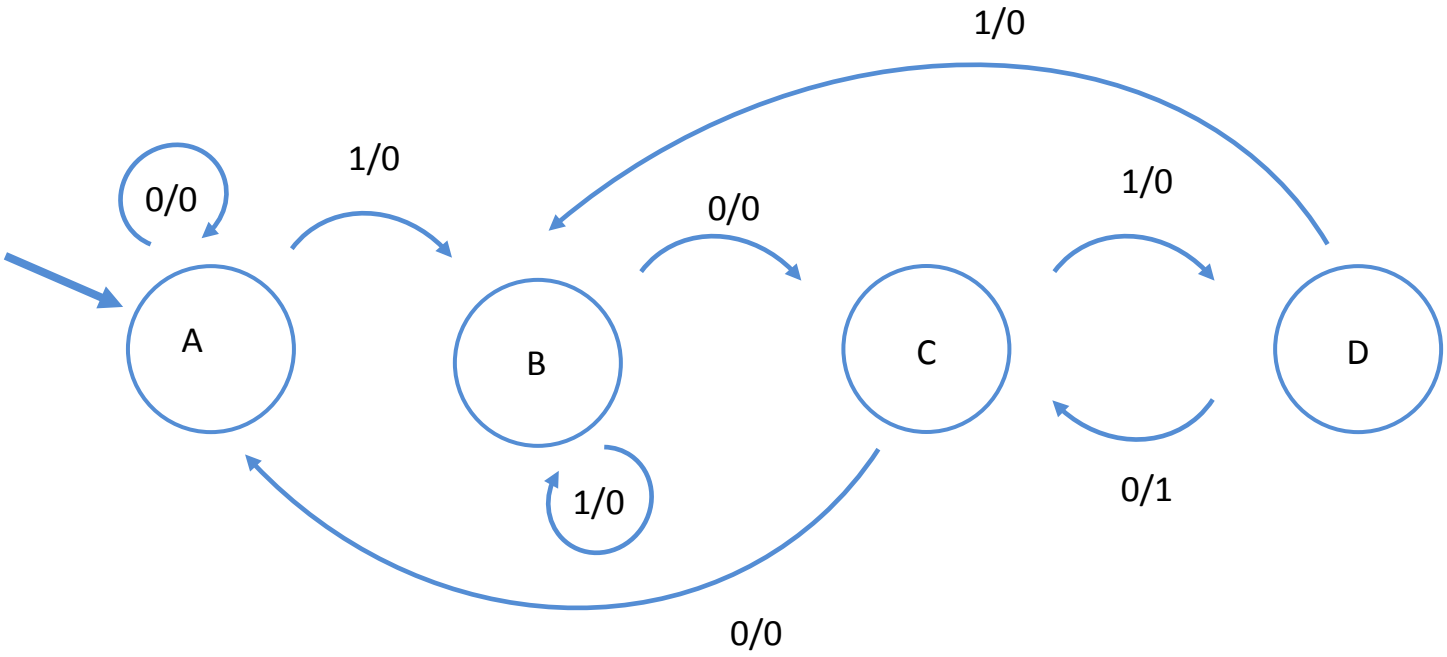
Purpose: To output a ‘1’ when the circuit sees a ‘1010’. Otherwise, output a ‘0’.

Example:

Input = 00110101000
Output = 00000010100

State Table

A = Nothing has been seen	00
B = A ‘1’ has been seen	01
C = A ‘10’ has been seen	10
D = A ‘101’ has been seen	11



Let's create our state tables for this:

Current and Next State Table (along with the Output)

Q1	Q0	Input	Q1+	Q0+	Out
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	0	1	0

D values required

D1	D0
0	0
0	1
1	0
0	1
0	0
1	1
1	0
0	1

T values required

T1	T0
0	0
0	1
1	1
0	0
1	0
0	1
0	1
1	0

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

D1 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	1	1	0
1	0	0	0	1

D1 = (Q0 & 'X) | (Q1 & 'Q0 & In)

D0 Generation Table (from Q1/Q0/Input values)

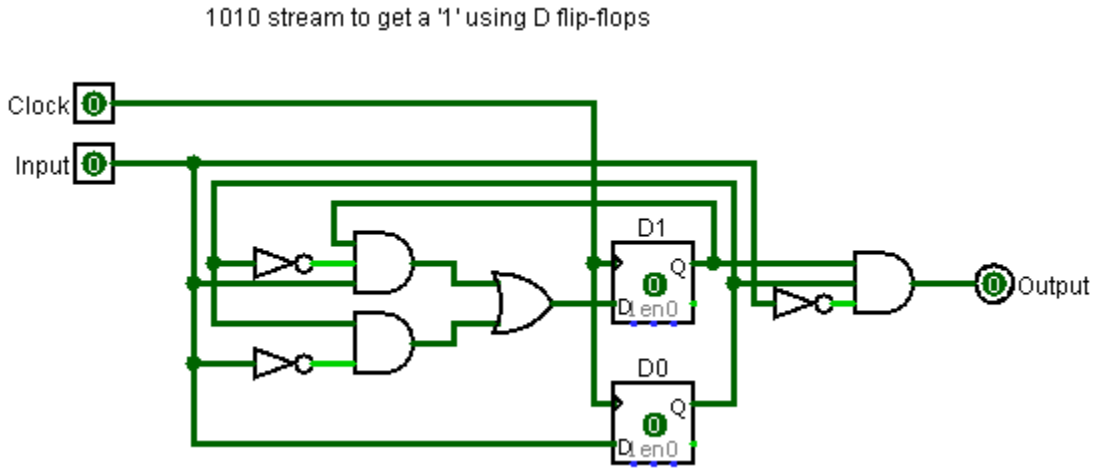
	Q1/Q0			
Input	00	01	11	10
0	0	0	0	0
1	1	1	1	1

D0 = In

Output Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	0	1	0
1	0	0	0	0

Out = Q1 & Q0 & 'In



Next, the 'T' flip-flop version.

T1 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	1	0	1
1	0	0	1	0

T1 = ('Q1 & Q0 & 'In) | (Q1 & Q0 & In) | (Q1 & 'Q0 & 'In)

T0 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	1	1	0
1	1	0	0	1

T0 = (Q0 & 'In) | ('Q0 & In)

Output Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
Input	00	01	11	10
0	0	0	1	0
1	0	0	0	0

Out = Q1 & Q0 & 'In

Well, that is one ugly bit of logic. I think I'll pass on creating THAT in Logisim. Your turn!

Verilog Version

We note pretty quickly that the output isn't just dependent on the state, but rather the state as well as input values, so it's a Mealy machine.

```
module sequence_recognizer (
	input	clk, in, reset,
	output reg [1:0] out
);

	reg		[1:0]state;			// Declare state register

	parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;	// Declare states

	always @ (posedge clk or posedge reset) begin
		if (reset)
			state <= S0;
		else
			case (state)
				S0:
					if (in) begin
						state <= S1;
					end
					else begin
						state <= S1;
					end
				S1:
					if (in) begin
						state <= S2;
					end
					else begin
						state <= S1;
					end
				S2:
					if (in) begin
						state <= S3;
					end
					else begin
						state <= S1;
					end
				S3:
					if (in) begin
						state <= S2;
					end
					else begin
```

```
                                        state <= S3;
                                end
                        endcase
        end

        always @ (state or in)
        begin
                        case (state)
                                S0:
                                        if (in) begin
                                                out = 2'b00;
                                        end
                                        else begin
                                                out = 2'b10;
                                        end
                                S1:
                                        if (in) begin
                                                out = 2'b01;
                                        end
                                        else begin
                                                out = 2'b00;
                                        end
                                S2:
                                        if (in) begin
                                                out = 2'b10;
                                        end
                                        else begin
                                                out = 2'b01;
                                        end
                                S3:
                                        if (in) begin
                                                out = 2'b11;
                                        end
                                        else begin
                                                out = 2'b00;
                                        end
                        endcase
        end

endmodule
```

From here, it should be easy to create a testbench for it.

Summary

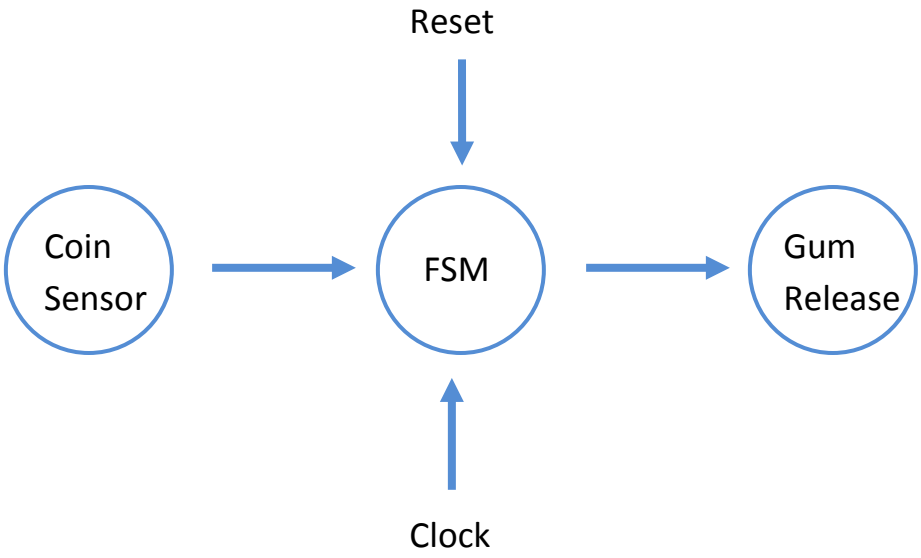
- Again, the State Machine Viewer doesn't represent this very well.
- This design was a few small edits away from our elevator design. This included widening the state register, increasing the number of states by 1 and making small modifications to the output as well as the next state logic.
- It was pretty easy to change.

Example 5: An Old Vending Machine

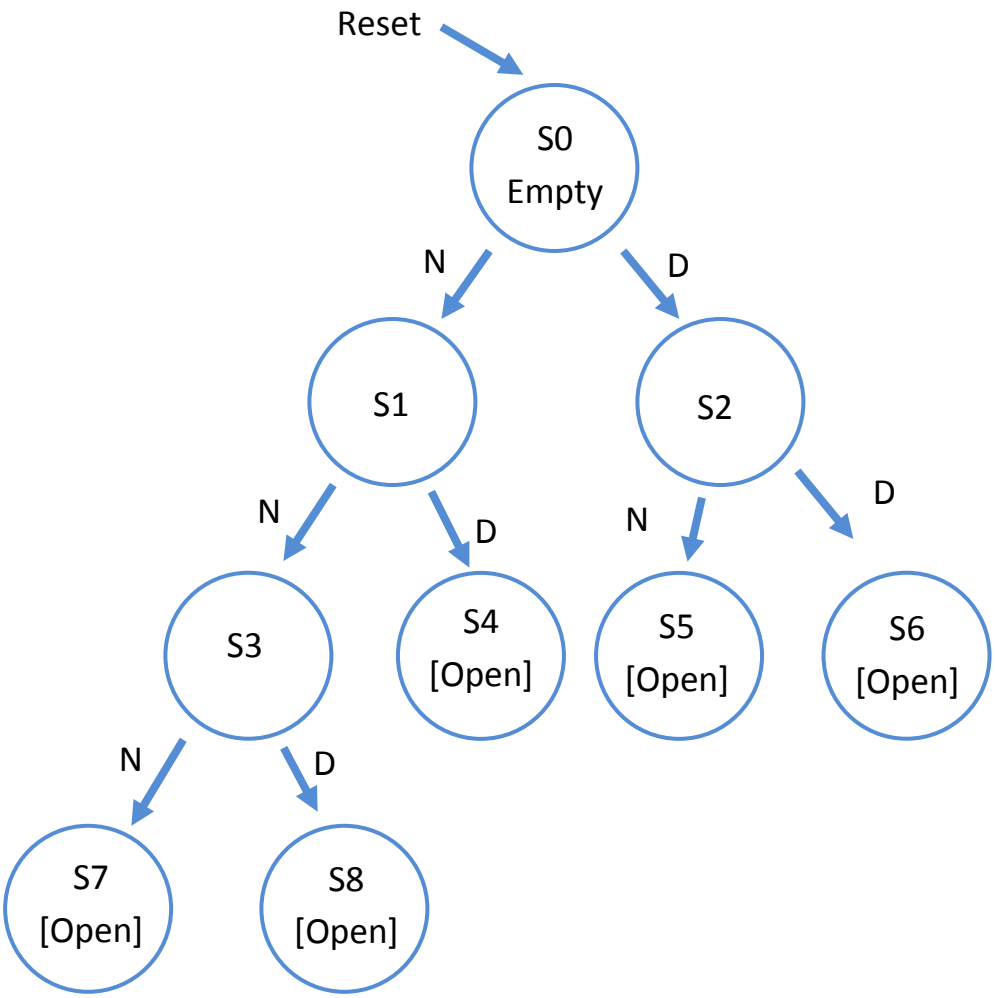
Requirements:

- Open the door for a package of Beeman's gum after 15 cents is deposited (for Chuck Yeager).
- Sorry, but no change is returned.
- A single slot for the money.

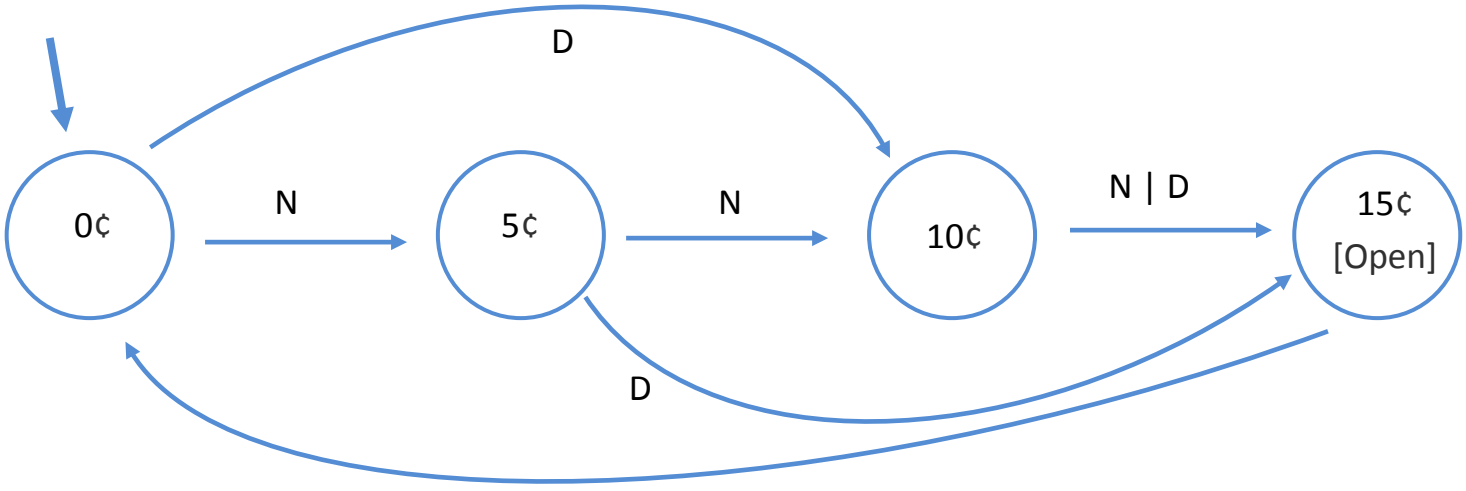
Let's draw it:



Our inputs are: • Nickel / Dime • Reset Our output is: • Open / Closed	Let's list the valid coin input sequences: • Three nickels • Nickel, dime • Dime, nickel • Two dimes • Two nickels, dime



Let's see if we can reduce those states:



Present State	Inputs		Next State	Output Open
	Dime	Nickel		
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	x	x
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	x	0
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	x	0
15¢	-	-	0¢	1

The ‘x’ is a not-valid state as you can’t insert two coins at once.

Let’s encode all this and then map it:

Current and Next State Table (along with the Output)

Q1	Q0	Dime	Nickel	Q1+	Q0+	Out
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	x	x	x
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
0	1	1	1	x	x	x
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	0	1	1	x	x	x
1	1	x	x	0	0	1

D values required

D1	D0
0	0
0	1
1	0
x	x
0	1
1	0
1	1
x	x
1	0
1	1
1	1
x	x
0	0

T values required

T1	T0
0	0
0	1
1	0
x	x
0	0
1	1
1	0
x	x
0	0
0	1
0	1
x	x
1	1

Q	Q+	D
0	0	0
0	1	1
1	0	0
1	1	1

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

D1 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
D/N	00	01	11	10
00	0	0	0	1
01	0	1	0	1
11	x	x	x	x
10	1	1	0	1

D1 = ('Q1 & D) | ('Q1 & Q0 & N) | (Q1 & 'Q0)

D0 Generation Table (from Q1/Q0/Input values)

	Q1/Q0			
D/N	00	01	11	10
00	0	1	0	0
01	1	0	0	1
11	x	x	x	x
10	0	1	0	1

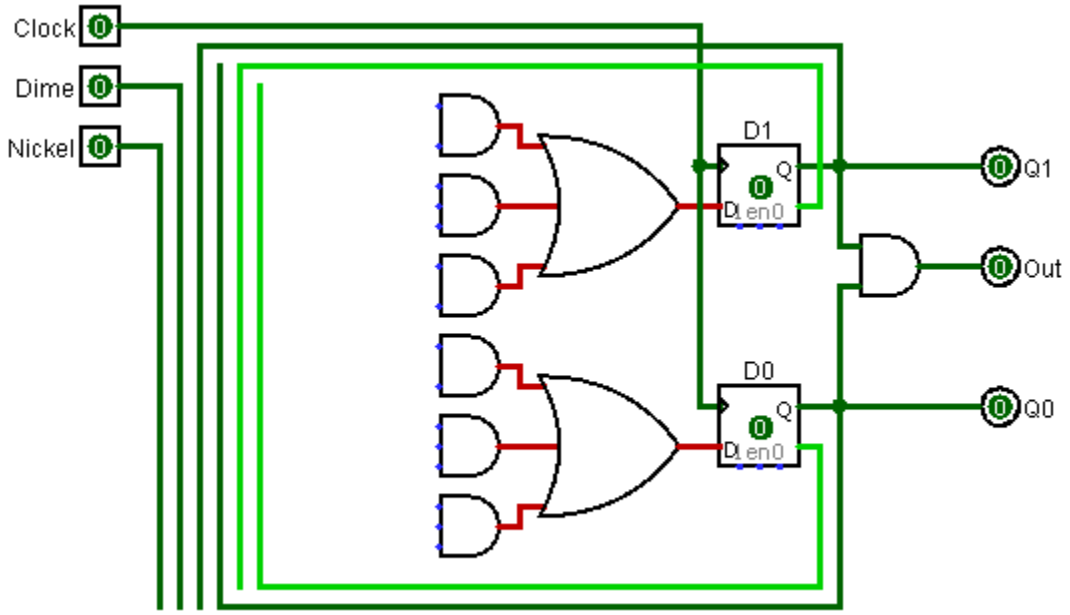
D0 = ('Q0 & N) | ('Q1 & Q0 & 'N) | (Q1 & 'Q0 & D)

Output Generation Table (from Q1/Q0/Input values)

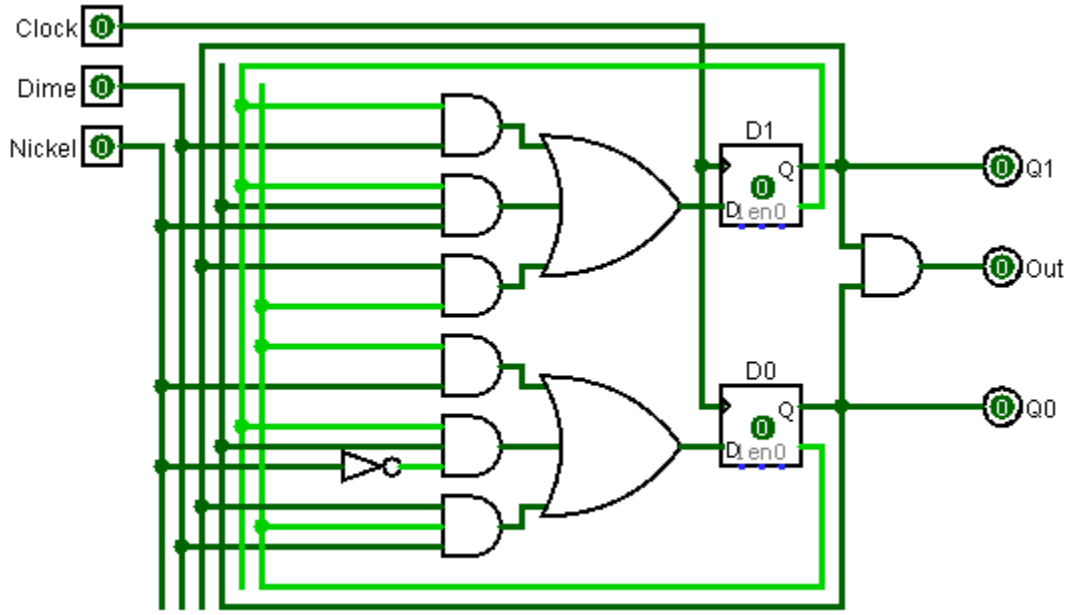
	Q1/Q0			
Input	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	x	x	x	x
10	0	0	1	0

Out = Q1 & Q0

We'll start out with this:



Here's our resultant 'D' flip-flop circuit.



Verilog Version

```
module vending_machine(
	input	clk, reset,
	input [1:0] in,
	output reg [0:0] out
);

	reg		[1:0]state;			// Declare state register

	parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;	// Declare states

	always @ (state) begin
		case (state)
			S0:
				out = 1'b0;
			S1:
				out = 1'b0;
			S2:
				out = 1'b0;
			S3:
				out = 1'b1;
		endcase
	end

	always @ (posedge clk or posedge reset) begin
		if (reset)
			state <= S0;
		else
			case (state)
			S0:
				if (in == 2'b01) begin
					state <= S1;
				if (in == 2'b10)
					state <= S2;
				end

			S1:
				if (in == 2'b01) begin
					state <= S2;
				if (in == 2'b10)
					state <= S3;
				end
				S2:
				if (in == 2'b01) begin
					state <= S3;
```

```
                                        if (in == 2'b10)
                                                state <= S3;
                                        end
                                        S3:
                                          state <= S0;
                                endcase

        end

endmodule
```

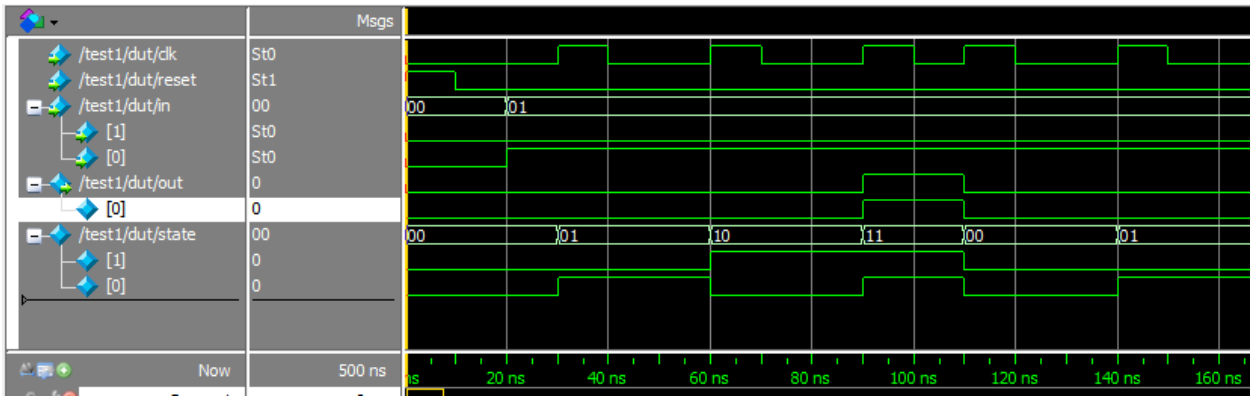
We need a testbench for that.

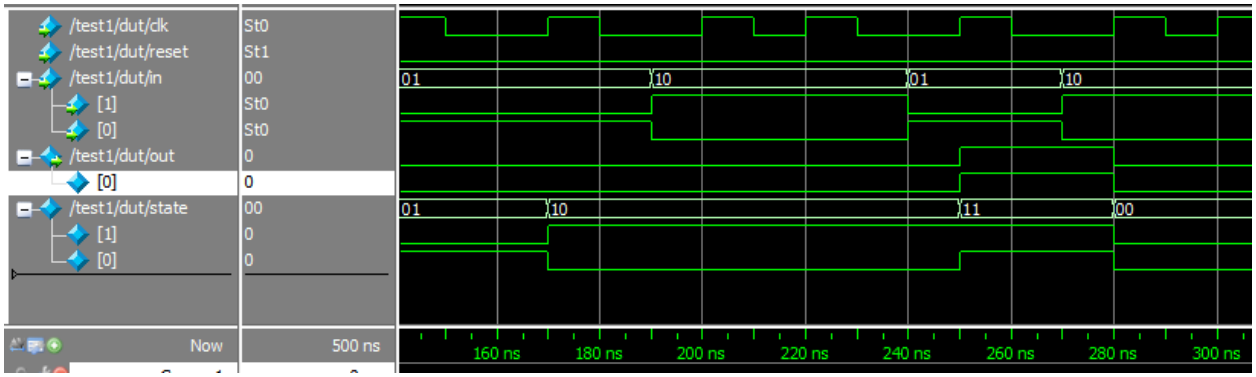
```
// Vending machine test bench
//

`timescale 1ns/1ns

module test1 ();

reg clk, reset;
reg [1:0] in;

vending_machine dut (clk, reset, in, out);

initial begin                                   // Here's the results from our waveform below
                clk=0; in=0; reset=1;   #10;            // 0ns - Clear everything and assert reset
                                                //      everything at 0ns. State, output = 0.
                reset = 0;              #10;            // 10ns - Clear reset .


        in = 2'b01;             #10;            // Start off with 3 nickels
        clk = 1;                        #10;
                clk = 0;                #10;
                in = 2'b01;             #10;
                clk = 1;                #10;
                clk = 0;                #10;
                in = 2'b01;             #10;
                clk = 1;                #10;
                clk = 0;                #10;
                clk = 1;                #10;
                clk = 0;                #10;



                in = 2'b01;             #10;            // Then 2 nickels and a dime
                clk = 1;                #10;
                clk = 0;                #10;
                in = 2'b01;             #10;
                clk = 1;                #10;
                clk = 0;                #10;
                in = 2'b10;             #10;
                clk = 1;                #10;
                clk = 0;                #10;
                clk = 1;                #10;
                clk = 0;                #10;


                in = 2'b01;             #10;            // Then 1 nickel and a dime
```

```
                clk = 1;            #10;
                clk = 0;            #10;
                in = 2'b10;         #10;
                clk = 1;            #10;
                clk = 0;            #10;
                clk = 1;            #10;
                clk = 0;            #10;


                in = 2'b10;         #10;        // Then 2 dimes
                clk = 1;            #10;
                clk = 0;            #10;
                in = 2'b10;         #10;
                clk = 1;            #10;
                clk = 0;            #10;
                clk = 1;            #10;
                clk = 0;            #10;


                in = 2'b10;         #10;        // Then 1 dime and a nickel
                clk = 1;            #10;
                clk = 0;            #10;
                in = 2'b01;         #10;
                clk = 1;            #10;
                clk = 0;            #10;
                clk = 1;            #10;
                clk = 0;            #10;

                                                // We should also try some with input = 0;

end

endmodule
```

ModelSim Waveforms

The waveforms are getting to the point, where you might want to add pass/fail text.





Summary

This is really a 4 state Moore machine with a 2 bit input. It's relatively straightforward when you think of it that way. The issue is that the testing can start to become exponentially more complicated as you increase the number of states, inputs and outputs to be tested.

Conclusion

This is obviously not an exhaustive study of Finite State Machines, but rather an overview including representation, analysis and implementation of a few simple examples.

For me, the next steps will be to learn various aspects of CPU architecture and implement one in Verilog.

Best of luck,

Andrew

References

- Digital Design and Computer Architecture by David Money Harris & Sarah L. Harris
- Digital Design Principles And Practices by John Wakerly
- Fundamentals of Logic Design by Charles H. Roth Jr. and Larry L. Kinney
- Logisim, available at http://sourceforge.net/projects/circuit/
- Altera Quartus II web package and ASE ModelSim, available at https://www.altera.com/download/sw/dnl-sw-index.jsp
- UC Berkely CS150 Course
- Bora Binary Explorer and the related ECED2200 course by Colin O'Flynn
- Google (of course)