Realizing a circuit from a State Diagram

What you've learned:

- What states are
- How many flip-flops/latches you need for a given number of states
- Boolean simplification using K-map or boolean algebra
- How to make a state diagram/finite state automaton
- How a latch works (as a black box)

What you're missing.

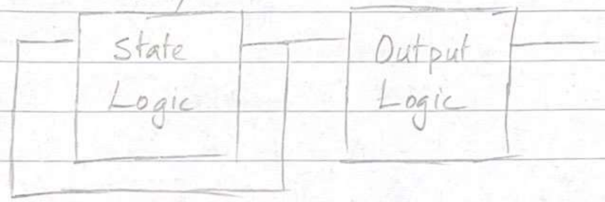
- How to put it all together
- Common state machine diagrams (Moore vs Mealy)

Starting with Moore vs Mealy

The Moore and Mealy FSM (finite state machine) are the two most basic forms of digital state machines. Moore needs more stuff as it defines the output state to be dependent on input and the current state. This is what you all are making for your final project. A Mealy FSM, on the other hand, only uses the current state to determine the output state.

Pictures!

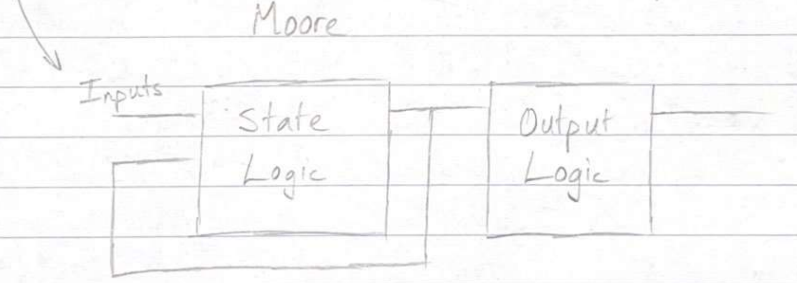
Mealy



The difference

Example: a counter or clock

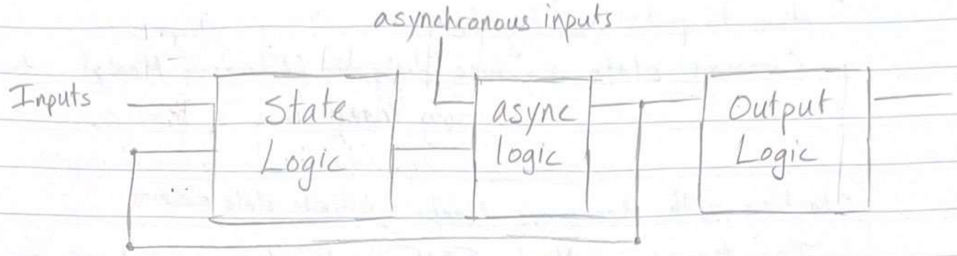
Moore



Example: a sensor controlled thermostat

If you're in the Wednesday lab, you'll notice there's a part missing. That would be the input logic, but that is because it is asynchronous. I actually made a mistake in the placement of this block, it should be right after state logic and before output logic as asynchronous logic should immediately change the outputs.

As such: Moore with asynchronous elements

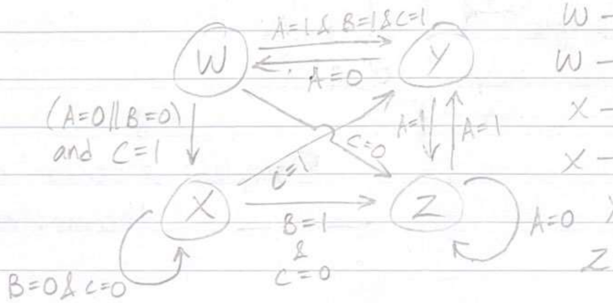


Since async logic is right before output logic, it can immediately update the output separate from the clock signal

Now that we have our parts, how do we build each part?
Let's start with State Logic:

With state logic, you typically have a state machine you want to implement. Between each state there is an input condition to reach a subsequent state. Then based on this, that means each subsequent state is a logic function of the inputs and the current state.

Example: Inputs A, B, C
States W, X, Y, Z



Transitions:
W → X if (A=0||B=0) &(C=1)
W → Z if C=0
W → Y if A=1 & B=1 & C=1
X → Z if B=1 & C=0 X → Y if C=1
X → X if B=0 & C=0
Y → Z if A=1 Y → W if A=0
Z → Y if A=1 Z → Z if A=0

async reset goes to Y.

This is a pretty simple state diagram since there are only 3 inputs, 4 states, and very small logic conditions for switching. But if you take a look at the Transition table, you can see how we can make equations out of it. For instance, transition $W \rightarrow Y$ if $A=1$ & $B=1$ is the same as $W = Y$ and $A=1$ and $B=1$ or $W=F(Y, A=1, B=1)$. A and B are inputs so the only part missing is what exactly are W and Y?

That's the next step: Enumerate your states

We have W, X, Y, and Z, so I'm just going to say $W=0$, $X=1$, $Y=2$, and $Z=3$. How many bits is that max... 2. That's why we need exactly 2 flip-flops/latches to hold 4 states. Say we had 14 states, to enumerate all states we would need 4 bits as 13 in binary (zero based counting) is 1101. So we would need 4 flip-flops/latches in that case.

Anyway, we now have values to plug in for W and Y

$$\overset{W}{[0,0]} = F(\overset{Y}{[1,0]}, A=1, B=1)$$

This is a bit weird to look at, so break it up for each bit of the state.

$$W0 = F_0(Y0, A=1, B=1)$$
$$W1 = F_1(Y1, A=1, B=1)$$

Based on the values for W0, W1, Y0, and Y1, we know F_0 must result in 0 if $A=1$ and $B=1$ and F_1 must also result in 0 if $A=1$ and $B=1$. However, that's only for 1 transition. What if A and $B \neq 1$? What should be the output of F_0 and F_1. Now we get to the transition truth table.

Having enumerated the states, we can use them as input bits for our Moore FSM.

2 output bits

I'm missing 2 rows. Ignore it for now.

	Old State		Inputs			New State		
	State Bit 1	State Bit 0	A	B	C	State Bit 1'	State Bit 0'	
	0	0	0	0	0	1	1	Z
	0	0	0	0	1	0	1	X
	0	0	0	1	0	1	1	Z
W –	0	0	0	1	1	0	1	X
	0	0	1	0	0	1	1	Z
	0	0	1	0	1	0	1	X
	0	0	1	1	0	1	1	Z
	0	0	1	1	1	1	0	Y
	0	1	0	0	0	0	1	X
	0	1	0	0	1	1	0	Y
	0	1	0	1	0	1	1	Z
X –	0	1	0	1	1	1	0	Y
	0	1	1	0	0	0	1	X
	0	1	1	0	1	1	0	Y
	0	1	1	1	0	1	1	Z
	0	1	1	1	1	1	0	Y
	1	0	0	0	0	0	0	W
	1	0	0	0	1	0	0	W
	1	0	0	1	0	0	0	W
Y –	1	0	0	1	1	0	0	W
	1	0	1	0	0	1	1	Z
	1	0	1	0	1	1	1	Z
	1	0	1	1	0	1	1	Z
	1	0	1	1	1	1	1	Z
	1	1	0	0	0	1	1	Z
	1	1	0	0	1	1	1	Z
	1	1	0	1	0	1	1	Z
Z –	1	1	0	1	1	1	1	Z
	1	1	1	0	0	1	0	Y
	1	1	1	0	1	1	0	Y

Now that's a really big truth table to work with. Let's shorten it. It's best to separate the output bits now as they are separate functions. Then find patterns where certain input bits don't matter for several consecutive rows and replace them with X into one row. For instance when the current state is 11, it doesn't matter what B and C are. If A=0, then the output bits are 11. Here's my shortening:

	S1	SO	A	B	C	S1'	
	0	0	X	X	0	1	$ABC+\bar{C}$
	0	0	0	0	1	0	
W	0	0	0	1	1	0	
	0	0	1	0	1	0	
	0	0	1	1	1	1	
	0	1	X	0	0	0	$B+C$
X	0	1	X	1	X	1	
	0	1	X	X	1	1	
Y	1	0	0	X	X	0	A
	1	0	1	X	X	1	
Z	1	1	X	X	X	1	← always 1 when S1 SO = 11

	S1	SO	A	B	C	SO'	
	0	0	0	X	X	1	$\bar{A}+\bar{B}+\bar{C}$ or $\overline{ABC}$
W	0	0	X	0	X	1	
	0	0	X	X	0	1	
	0	0	1	1	1	0	
X	0	1	X	X	0	1	$\bar{C}$
	0	1	X	X	1	0	
Y	1	0	0	X	X	0	A
	1	0	1	X	X	1	
Z	1	1	0	X	X	1	$\bar{A}$
	1	1	1	X	X	0	

This makes it a little easier to make correlations as seen above. This is typically only possible if transition conditions are small. You'll also find a direct correlation between the output function and the transition conditions.

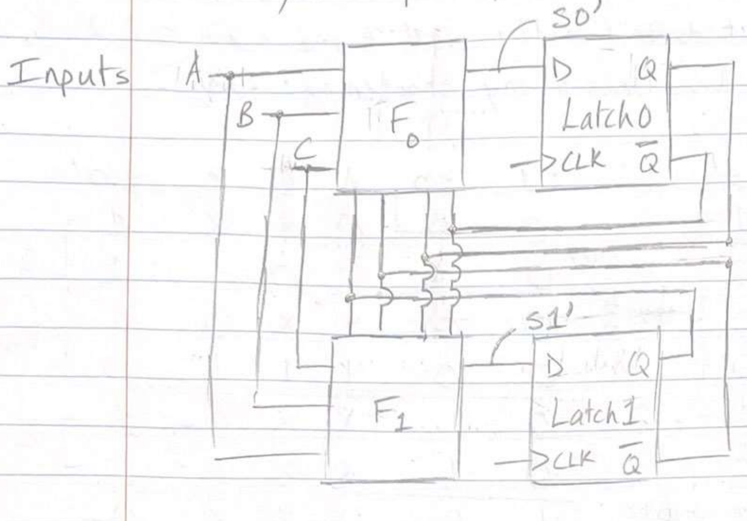
If we write the full equations down for S1' and SO', we get

$$S1' = \overset{0}{\overline{S1}\,\overline{SO}}(ABC+\bar{C}) + \overset{1}{\overline{S1}\,SO}(B+C) + \overset{2}{S1\,\overline{SO}}A + \overset{3}{S1\,SO}$$

$$SO' = \overline{S1}\,\overline{SO}(\bar{A}+\bar{B}+\bar{C}) + \overline{S1}\,SO(\bar{C}) + S1\,\overline{SO}A + S1\,SO(\bar{A})$$

You may choose to simplify this system as you wish with some possibilities in the other post.

I'm not going to draw the logic out, instead, I will represent the logic function for S0' as F0 and S1' as F1 because I'm lazy. Where do these go? F0 and F1 make up the State Logic right before your flip-flop/latches



Note: Why pass in $\overline{Q}$? Latch 0 holds state bit 0 aka S0. Then $\overline{Q}$ must be $\overline{S0}$. You can remove two NOT gates this way

What exactly is going on here?

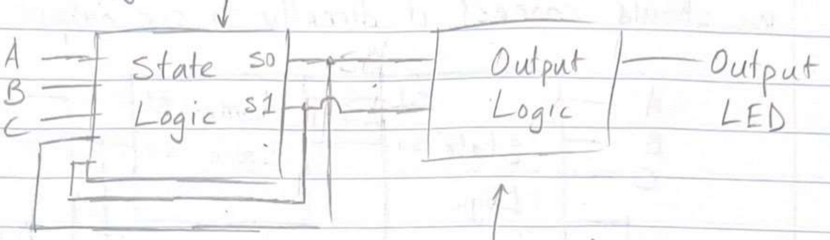
S0, $\overline{S0}$, S1, and $\overline{S1}$ are being passed into black boxes F_0 and F_1 defined on the previous page with inputs A, B, C. Based on the output of the black boxes, the new state is determined. However, D latches won't change their value (update their state) until the next clock edge. So where does the value go? It stays on the wire between the black box and the latch until the clock edge. Why though? A logic function will not change its output unless the input values change. You know S0 and S1 won't change between clock edges because it is held by the D latch. Then you can also assume digital logic is so fast that a change in A, B, or C will finish much faster than a clock edge period (Digital logic picosecond scale, clock is on millisecond scale; button press is on 0.1 second scale). So S0' and S1' will be immediately ready and essentially constant as soon as a button is pressed and will remain the same until the next clock edge to change the state.

We will get to reset soon

That's all for State Logic

Now that the hard part is done, let's do the easy part: Output Logic

You have a bigger black box now and let's assume it works



We need to figure this part out now

First, what are the inputs? S1 and S0, our state bits. So, let's start by defining what the output should be for each state. I'm also going to use RGB LED because it makes sense.

Let's pick colors first: W = white X = red Y = yellow Z = off

RGB has 3 pins, Red, Green, and Blue

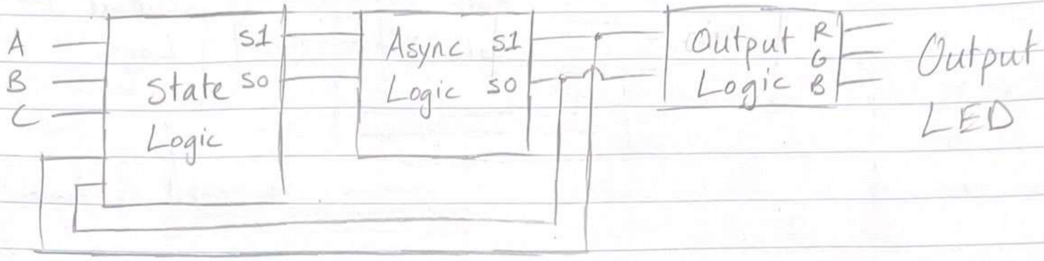
	R	G	B
State	Pin1	Pin2	Pin3
W	1	1	1
X	1	0	0
Y	1	1	0
Z	0	0	0

Let's expand this to state bits

S1	S0	Pin1	Pin2	Pin3
0	0	1	1	1
0	1	1	0	0
1	0	1	1	0
1	1	0	0	0

Now you just need to make a logic function for each pin based on S0 and S1. I trust you all can do that so I'm moving on.

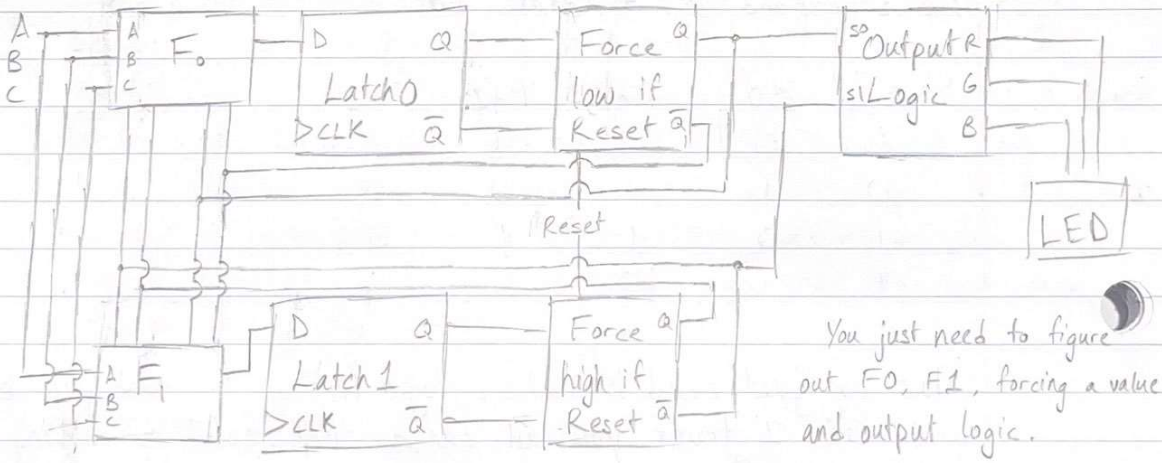
With a complete State Logic and Output Logic, you've technically built an entire system. But you need one last element for this project, the asynchronous element. This should update your output immediately. Then we should connect it directly to our output like so:



An asynchronous element will typically force a single state as it would mainly be used in the case of errors, exceptions, or malfunctions. Everything else should be handled by the state pattern.

If we are going to force a single state if a signal is high (or low depending on your system) and keep the original state otherwise, then what can we use? Keep in mind the properties of gates, that you have access to VCC and GND pins, and recently learned topics. At this point you should have guessed it; it's quantum mechanics! Just kidding, but I'm going to let y'all decide what you want to do here.

That's it for all the logic. Let's look inside a full implementation



The force low and force high here are to create state Y as defined in the transition table earlier

$Y = 1\ 0$

S1 S0

You just need to figure out F0, F1, forcing a value, and output logic.