University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Machine Models

FSM Outputs May or May Not Depend Directly on Inputs

As mentioned previously, for our class,
FSM outputs depend only on state,
not on FSM inputs.

Historically, such an FSM was called
a **Moore** machine.

The more general model,

- in which outputs may also depend on inputs,
- was called a **Mealy** machine.

More General Model Always Used in Practice

In practice,

- designers always use Mealy machines,
- so **FSM outputs may depend directly on inputs**.

If a designer wants

- an output to be independent of inputs,
- the designer simply designs the FSM to meet that requirement.

So the names are just of historical interest.

Inputs May Allow Us to Design a Smaller FSM

Why use the general model?

Inputs carry information.

We can sometimes build a smaller FSM
if we make use of that information.

More General Model Can Introduce Timing Issues

Why do we use the simpler model in ECE120?

If outputs depend directly on inputs,

output timing also depends on input timing,

so we lose the benefit of treating time as a discrete value (an integer).

An Example Illustrates the Tradeoffs

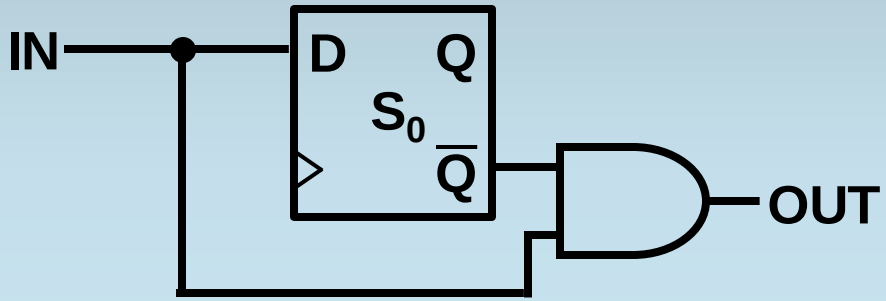
Let's use an example to illustrate these tradeoffs.

Say that we want to recognize the sequence **01** in a serial input **B**.

Whenever **B** is **0** in one cycle and **1** in the next cycle, we set the output **Z** equal to **1**.

Mealy Machine for 01 Sequence Recognizer

Consider the design to the right.



What is the next-state equation ($S_0^+ = ?$)?

$$S_0^+ = IN$$

Current IN is 1.

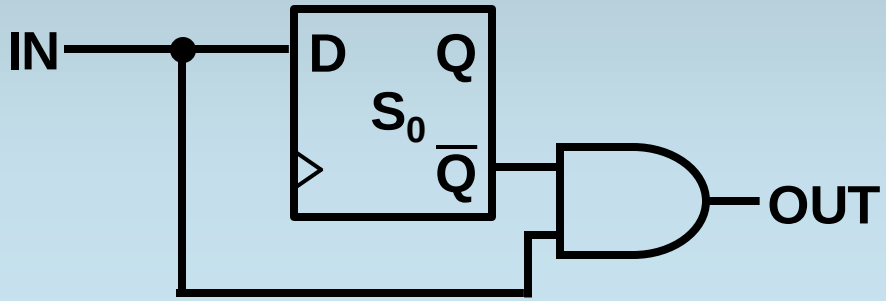
And the output equation?

$$OUT = IN \cdot S_0'$$

Last IN was 0.

Transition Diagrams Look Different for Mealy Machines

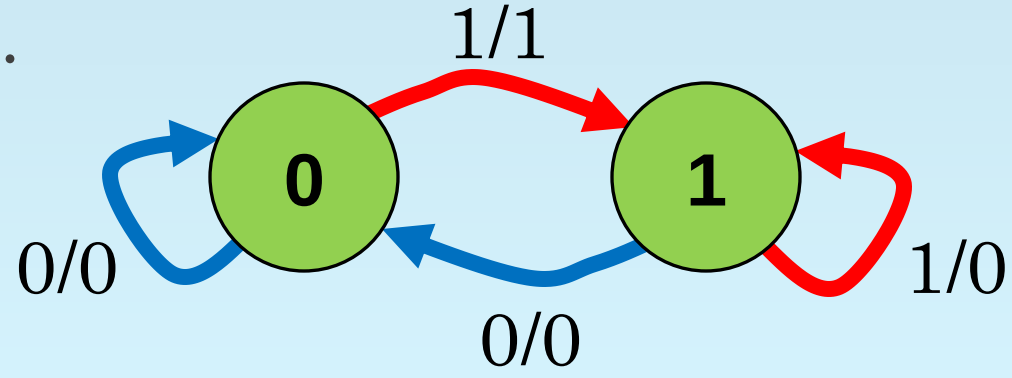
Now let's draw the state diagram.



We have two states.

Outputs depend on inputs, so

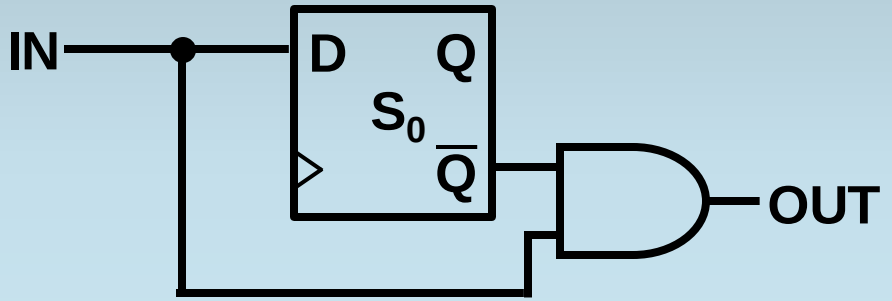
- states cannot be labeled with outputs.
- Instead, transitions are labeled with outputs.



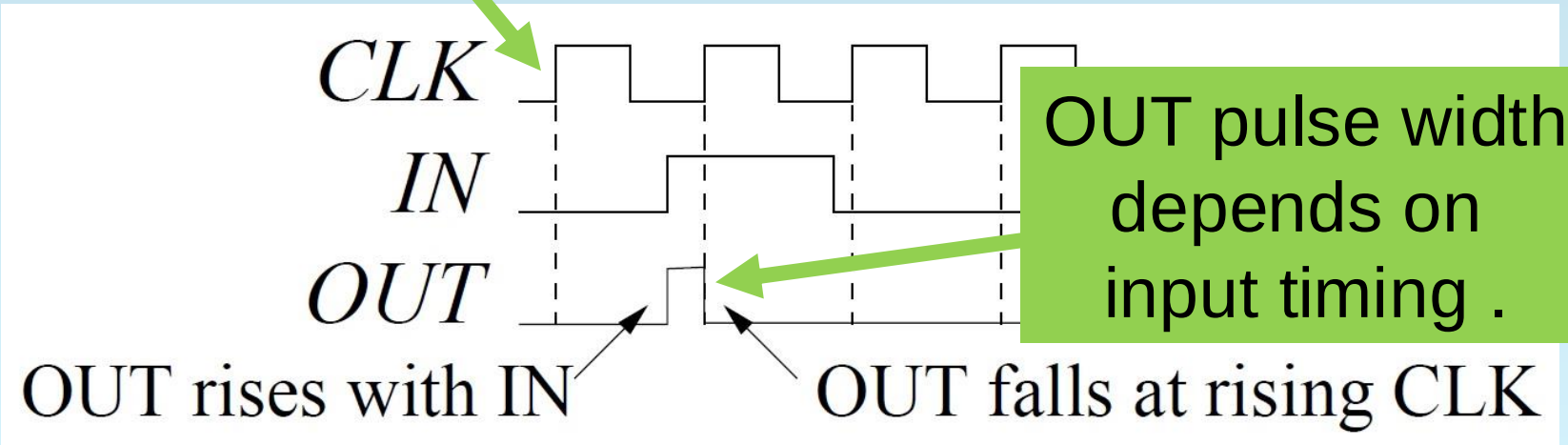
A Timing Diagram Reveals the Timing Issues

Recall that

OUT = IN · S_0'



S_0 is 0 after this edge.



We Can Usually Ignore the Narrow Output Problem

Usually, narrow output pulses don't matter.

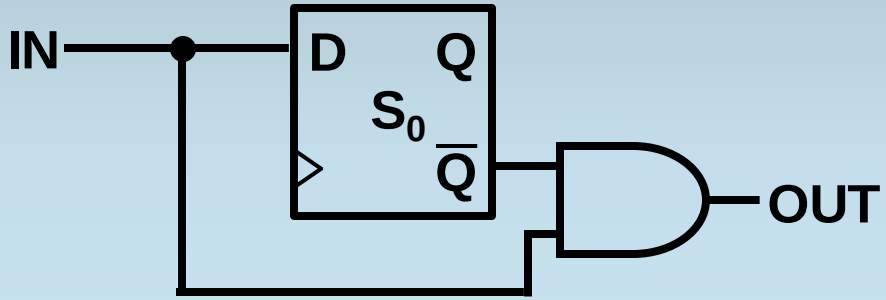
If inputs

- come from flip-flops on the same clock,
- changes arrive early enough (but may limit clock speed).

We may have problems **if inputs are external or if outputs are used externally** (not on the same clock).

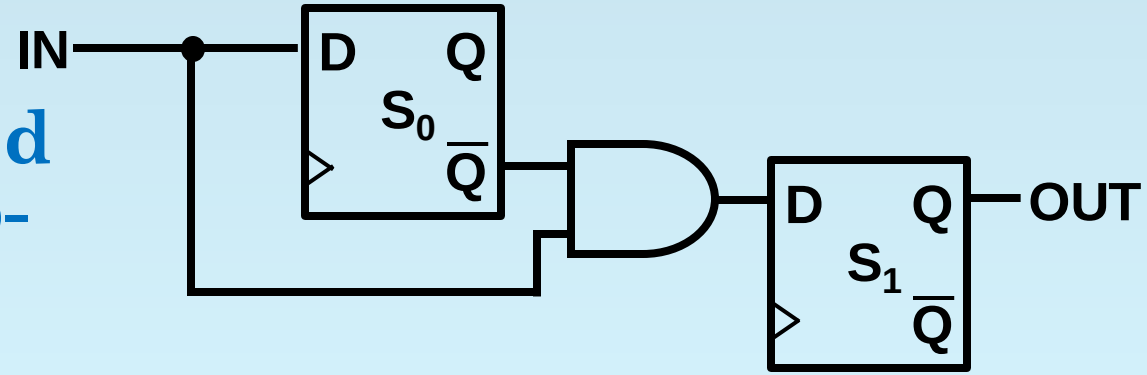
How Can We Fix the Narrow Pulse Problem?

What if we need a wider output pulse?



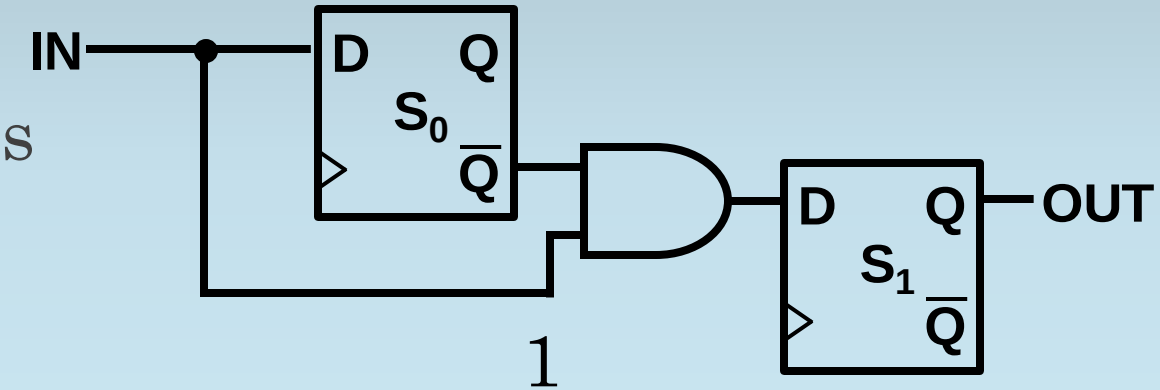
Do we have to redesign the system entirely? (as a Moore machine)

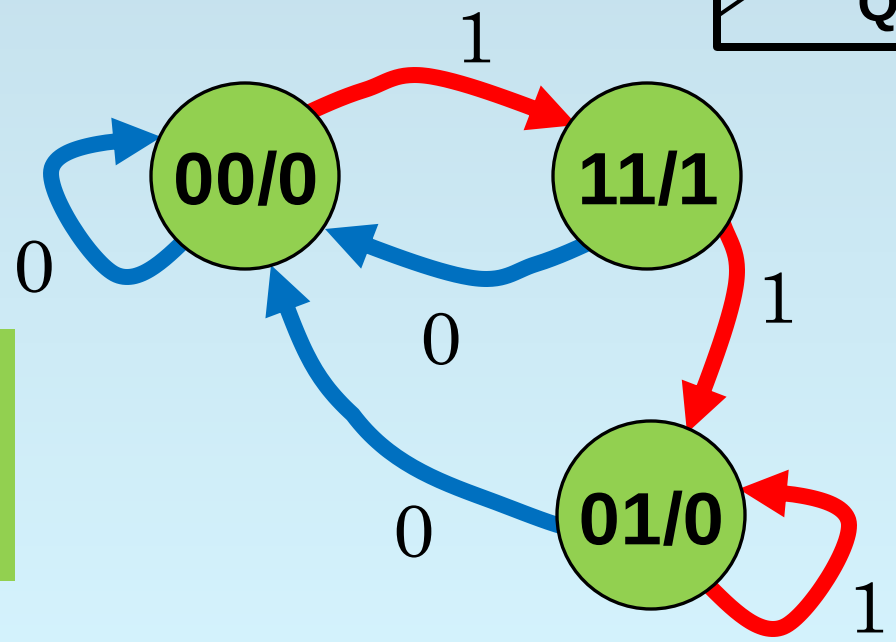
No! Just add another flip-flop.



The New Flip-Flop Splits One State into Two

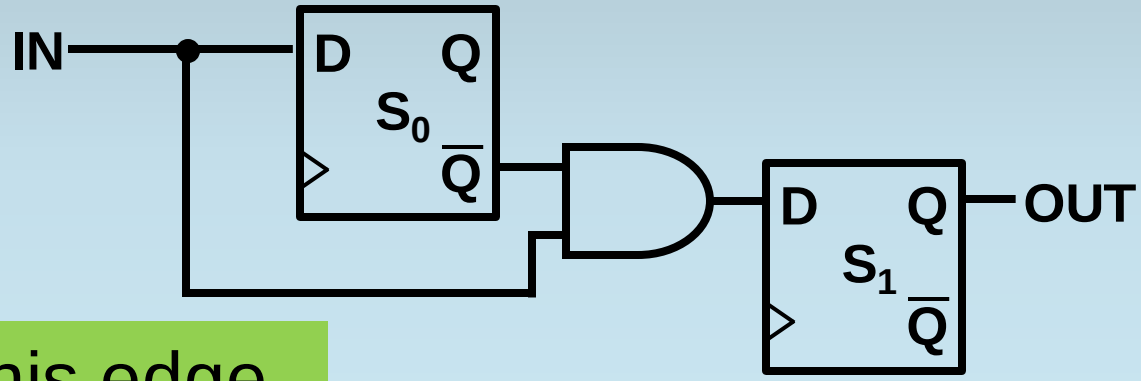
The new flip-flop splits the "1" state into "11" and "01" states.



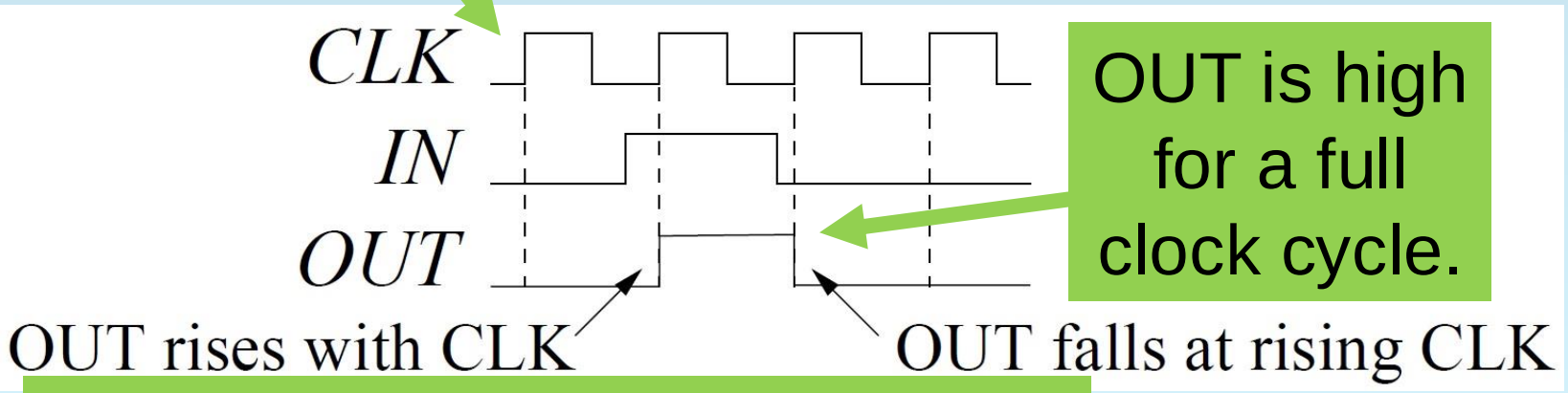


Note: the 01 state is not reachable from other states.

OUT is Delayed but High for a Full Cycle with Moore



S_0 is 0 after this edge.



(OUT=1 delayed to rising edge.)

Summary of the Two Models

Moore: outputs depend only on state, not on inputs

Mealy: outputs can also depend on inputs

Mealy is used in practice.

- Can reduce size of design, but
- may create thin output pulses.

Solving these problems is easy:
add flip-flops to make a Moore design.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

FSM Design Process

We Follow a Six-Step Process to Design an FSM

Here is a structured methodology that you can use to develop an FSM.

There are six steps:

1. develop an abstract model
2. specify I/O behavior
3. complete the specification
4. choose a state representation
5. calculate logic expressions
6. implement with flip-flops and gates

Step 1: Develop an Abstract Model

First, we translate our ideas and thoughts
- from human language
- into a model with states and desired behavior.

For now, just capture intended use
(no need to be thorough nor complete).

What are the different states of the system?

How do we expect it to move amongst these states?

Step 2: Specify I/O Behavior

Start to formalize a little by specifying input and output behavior.

Input and output must consist of bits.

How many inputs are needed?

What representation is used?

And the same questions for outputs.

Sometimes, the FSM I/O must match other systems, so representations (using bits) are already defined.

Step 3: Complete the Specification

We are now ready to resolve any ambiguities

- by making design decisions
- (in other words, choosing behavior).

Implicit assumptions should also be made clear and written down.

We may choose to leave some behavior as "don't care," but such a decision should be made carefully (and checked later).

Step 4: Choose a State Representation

The FSM state representation will affect logic for both next states and outputs.

Some ways to choose

- match state to output (output patterns must be unique),
- map states to hypercube such that transitions are mostly along edges, or
- use human meaning for state bits.

The last is a good way to choose because it separates bits into meaningful groups that may not affect each other (thus simplifying logic).

Step 5: Calculate Logic Expressions

Once you have completed the specification of state IDs, next states, and outputs in bits, all that's left is to build combinational logic.

If you have a lot of variables, breaking the truth tables up may help.

State bits that have human meaning also helps to simplify here: bits may be ignored if they are not relevant.

Step 6: Implement with Flip-Flops and Gates

State bits are stored in flip-flops.*

Next-state and output logic are built in the same way that you build any other combinational logic.

There's nothing special about it.

Hook the next-state logic outputs to the D inputs of the flip-flops.

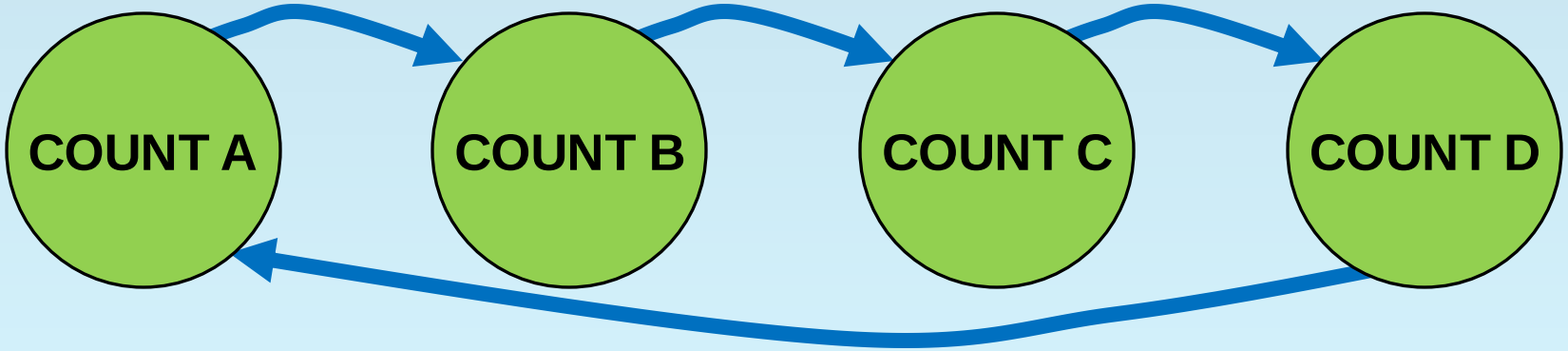
Output bits are functions of the flip-flop state.

*Registers, shift registers, and counters are fine, too. We'll use those in a week or so.

A Quick Example: A 2-Bit Gray Code Counter

Let's design a 2-bit Gray code counter using our methodology.

1. Our abstract model? A counter that goes through four states. Like this:

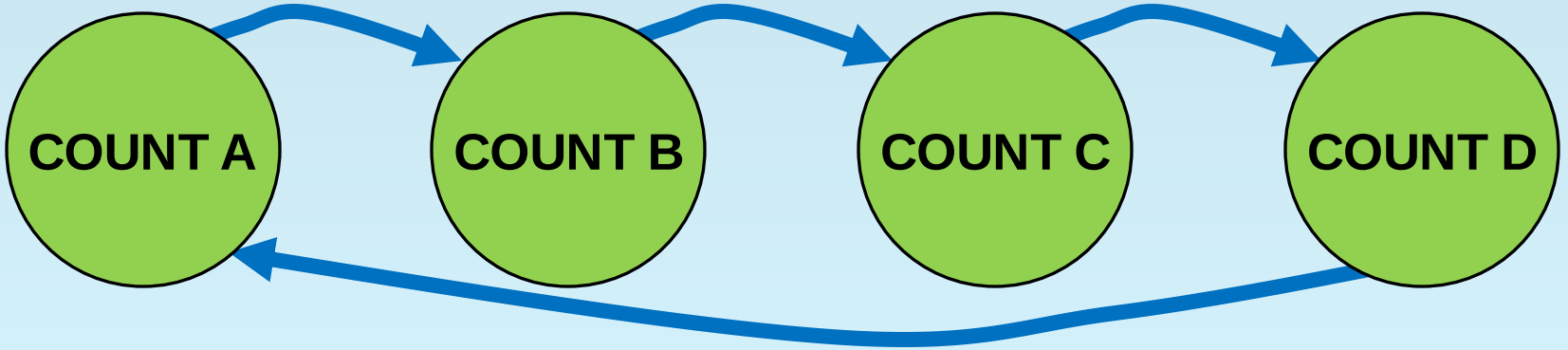


Defining I/O and Completing the Specification

2. Inputs: none (it's a counter).

   Outputs? 00, then 01, then 11, then 10, then back to 00.

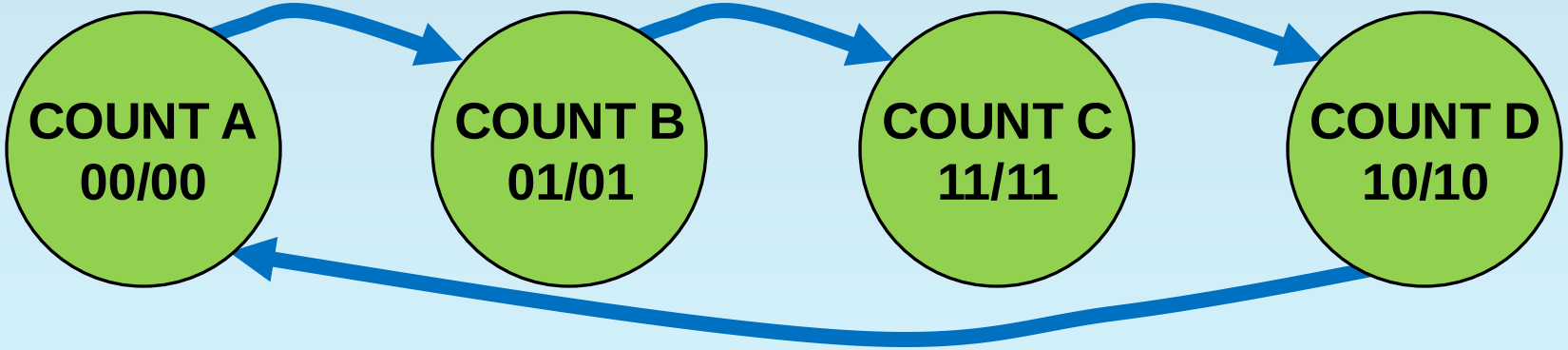
3. No inputs, so … specification is complete!



Choose Output Bits as the State IDs

4. What about the representation?

   The outputs are unique, so let's use them as state IDs as well. Then we need no output logic.



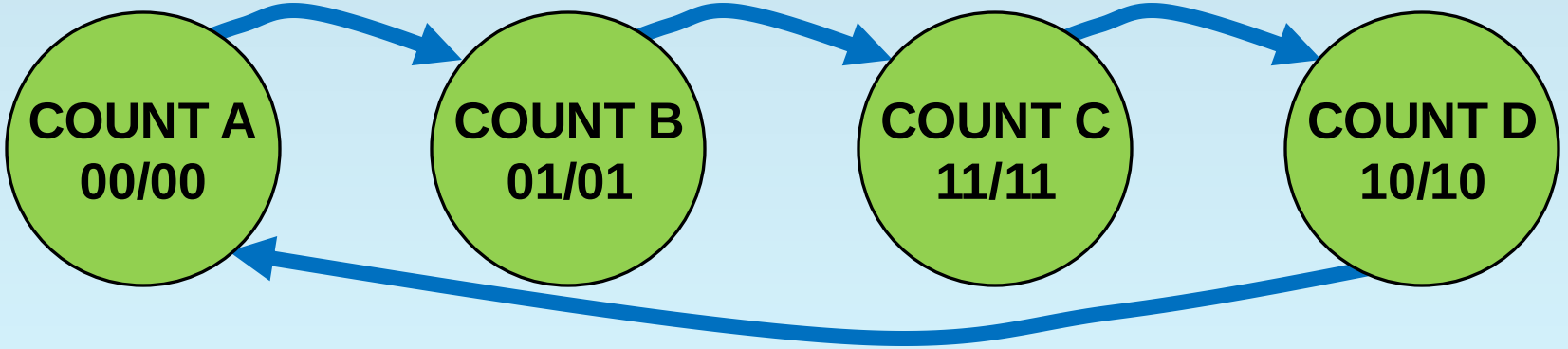
Solve the Next-State Equations

5. Now we can write equations from a truth table.

$S_1^+ = S_0$

$S_0^+ = S_1'$

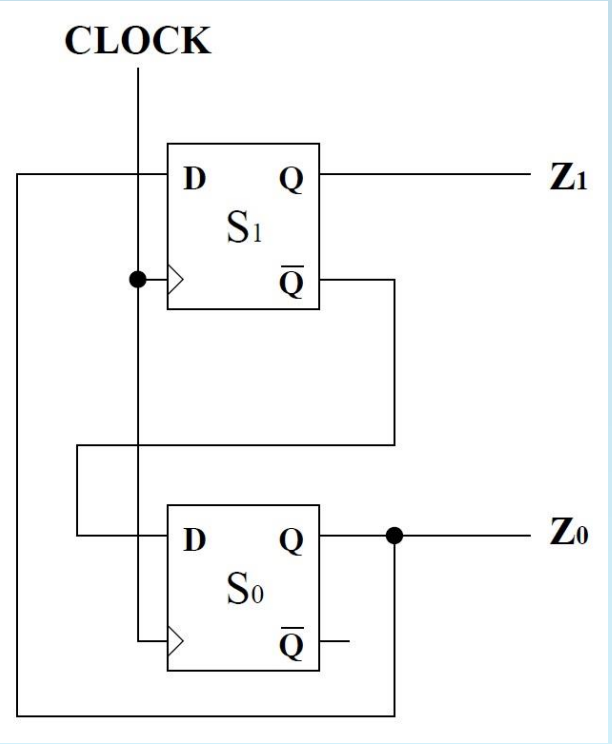
S_1	S_0	S_1^+	S_0^+
0	0	0	1
0	1	1	1
1	0	0	0
1	1	1	0



Implement Using Two Flip-Flops

6. Finally, we can implement, as shown below.

$$S_1^+ = S_0$$
$$S_0^+ = S_1'$$



University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

A Color Sequencer

Review the Six-Step Process

Recall our six-step process for FSM Design:

1. develop an abstract model
2. specify I/O behavior
3. complete the specification
4. choose a state representation
5. calculate logic expressions
6. implement with flip-flops and gates

Let's Build a Color Sequencer

Let's do another example.

Let's build a color sequencer that cycles through a set of colors.

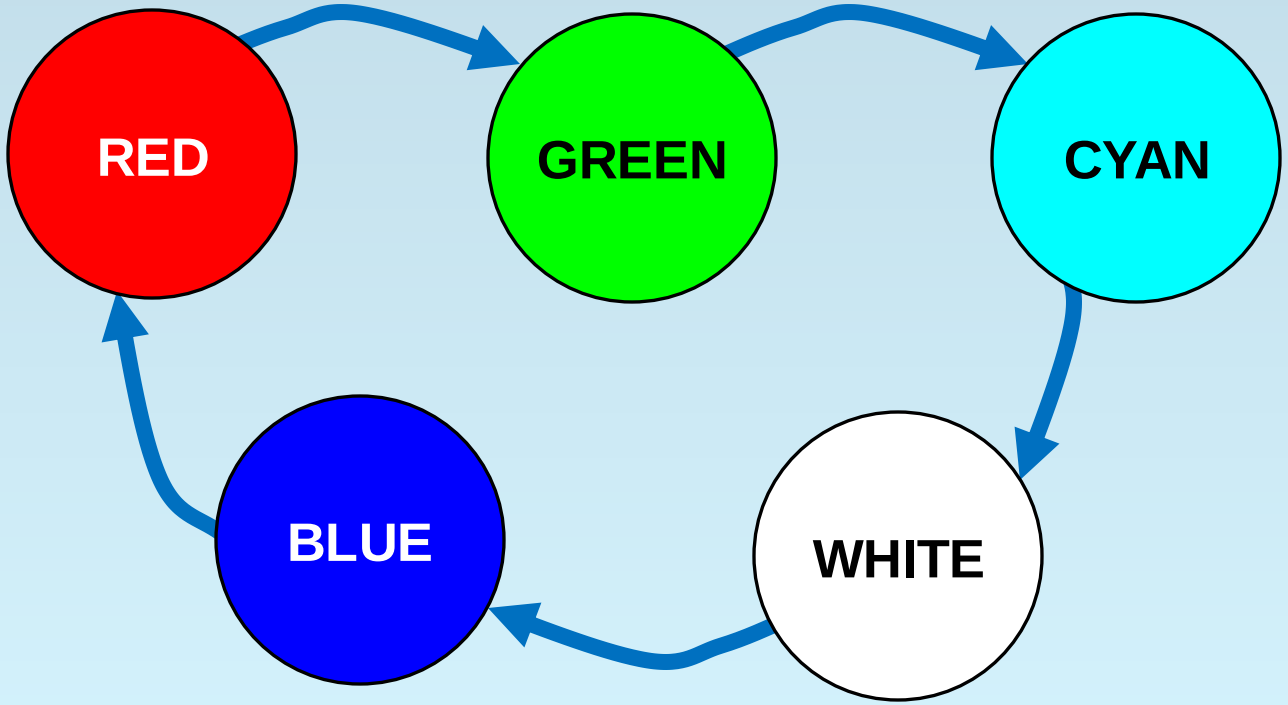
Imagine that we have an LED light that can output eight colors…

Our FSM will drive this light using the RGB signals.

RGB	color
000	black
001	blue
010	green
011	cyan
100	red
101	violet
110	yellow
111	white

Abstract Model for a Color Sequencer Has Five States

1. Our abstract model? A counter that goes through five colors. Like this:



Next, Define Inputs and Outputs

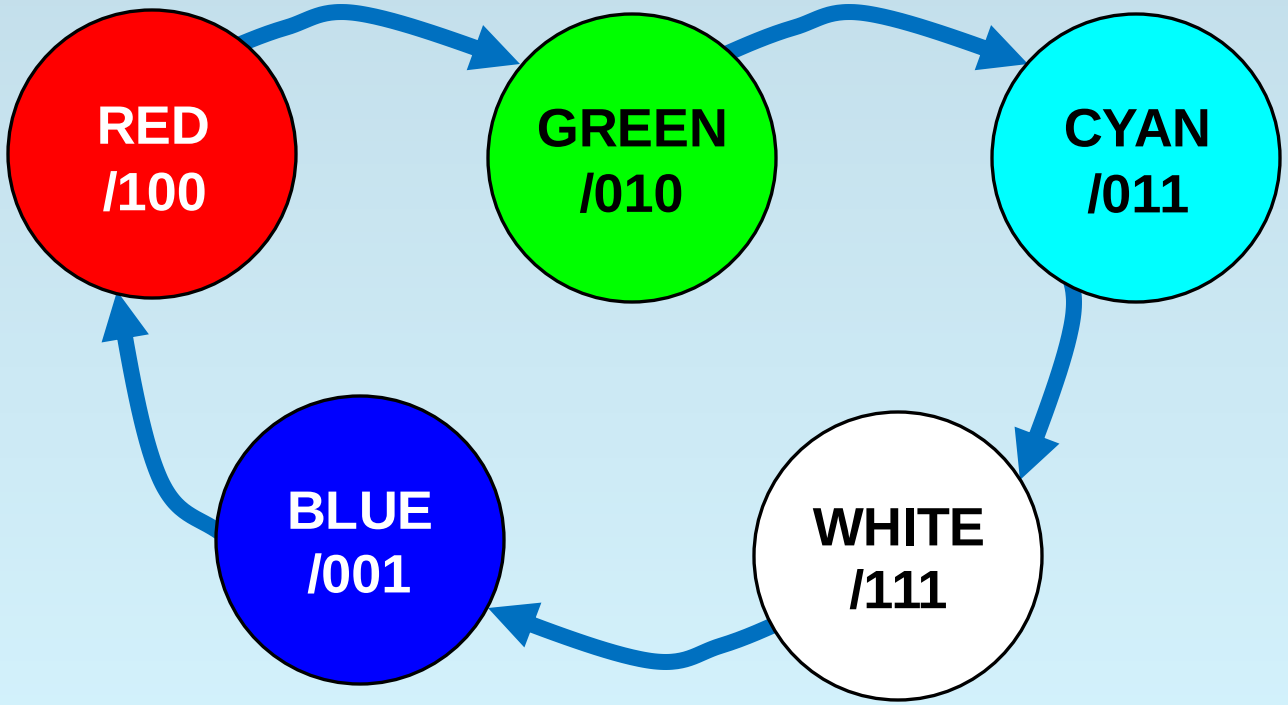
2. Inputs: none (it's a counter).

   Outputs? We can just read RGB from the table for each state.

RGB	color
000	black
001	blue
010	green
011	cyan
100	red
101	violet
110	yellow
111	white

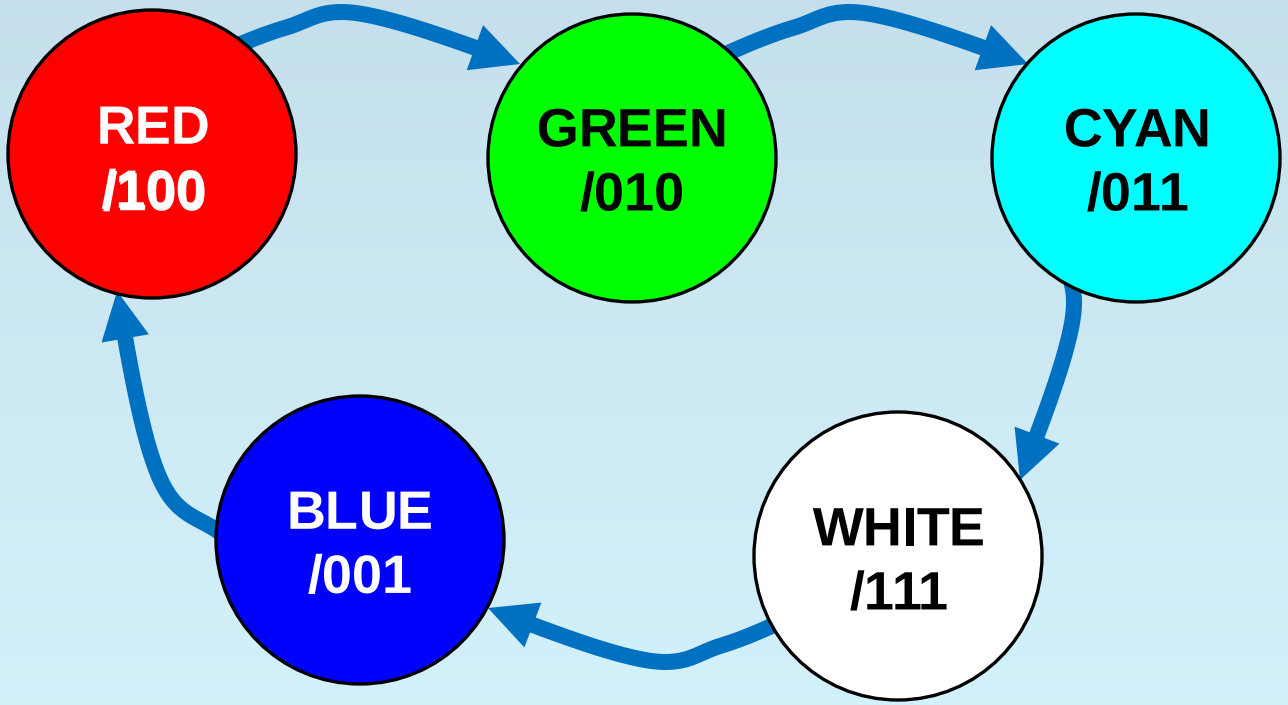
Outputs Represent Red, Green, and Blue

Let's add the outputs (as /RGB) to the states.



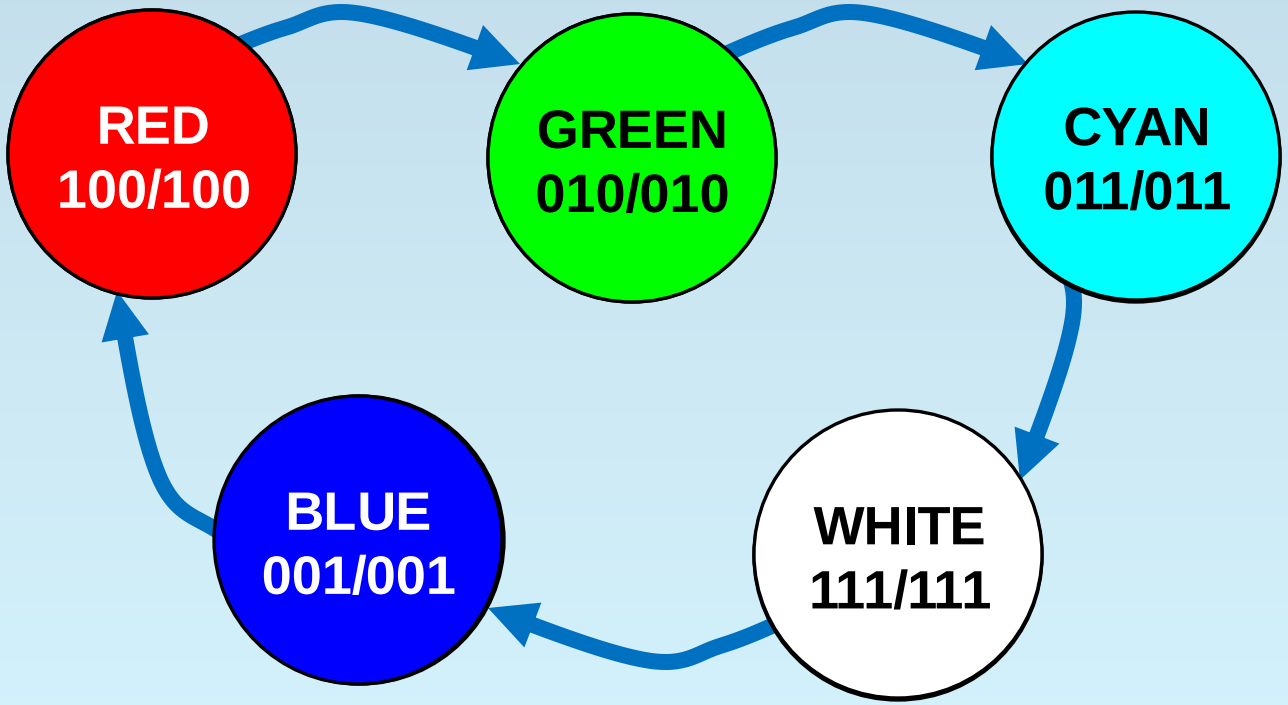
Completing the Specification

3. No inputs, so … specification is complete!



Use Unique Outputs as the Internal State IDs

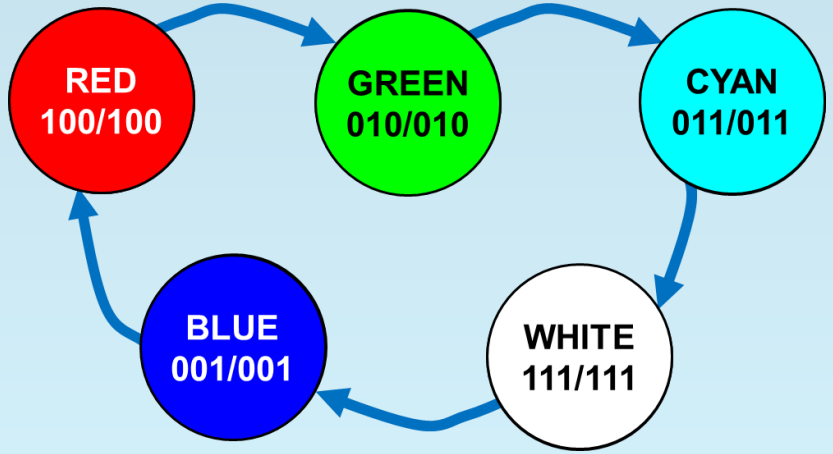
4. Outputs are again unique, so use them as state IDs as well.



Write a Next-State Table

S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	x	x	x
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	x	x	x
1	1	0	x	x	x
1	1	1	0	0	1

5. Time for equations. Start by writing a next-state table.



Now Use K-Maps to Express the Next-State Values

S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	x	x	x
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	x	x	x
1	1	0	x	x	x
1	1	1	0	0	1

Now copy into K-maps.

$$S_0^+ = S_1$$

S_0^+

S_2 \ S_1S_0	00	01	11	10
0	x	0	1	1
1	0	x	1	x

Now Use K-Maps to Express the Next-State Values

S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	x	x	x
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	x	x	x
1	1	0	x	x	x
1	1	1	0	0	1

Now copy into K-maps.

$$S_1^+ = S_2'S_1 + S_2S_1'$$
$$= S_2 \oplus S_1$$

S_1^+

S_2 \ S_1S_0	00	01	11	10
0	x	0	1	1
1	1	x	0	x

Now Use K-Maps to Express the Next-State Values

S_2	S_1	S_0	S_2^+	S_1^+	S_0^+
0	0	0	x	x	x
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	1	x	x	x
1	1	0	x	x	x
1	1	1	0	0	1

Now copy into K-maps.

$$S_2^+ = S_2'S_0 = (S_2 + S_0')'$$

S_2^+

S_2 \ S_1S_0	00	01	11	10
0	x	1	1	0
1	0	x	0	x

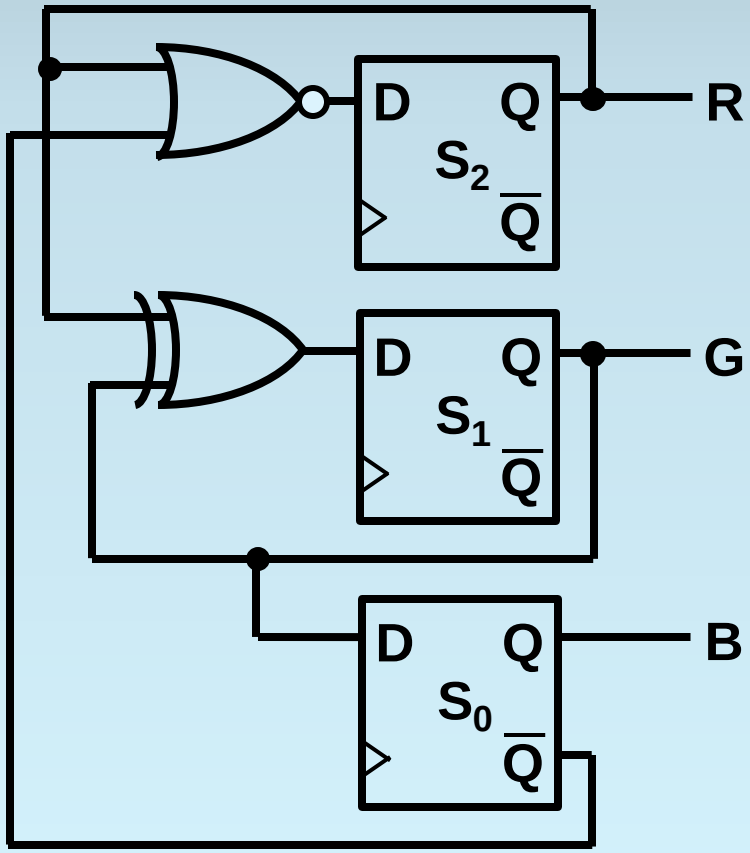
Implement Using Three Flip-Flops and Two Gates

6. Finally, we can implement, as shown to the right.

$$S_2^+ = (S_2 + S_0')'$$

$$S_1^+ = S_2 \oplus S_1$$

$$S_0^+ = S_1$$



Ready to Build It?

Are you excited?

Imagine that you go get your protoboard out.

You go to the lab.

You build the color sequencer.

You hook it to the LED light.

You turn it on.

…

It stays black.

Seem familiar?

Behavior Seems to Be Inconsistent

You debug for a while.

You play with wires.

You look at datasheets.

Everything seems right.

Sometimes it works.

Sometimes it flashes yellow or violet, then works.

Sometimes it stays black.

What's going on?

Our Don't Cares Become 0s for S_2

What happened to the "don't care" states?

Let's take a look.

We can use our K-maps or our equations.

For S_2, the x's became 0s.

S_2^+

S_2 \ S_1S_0	00	01	11	10
0	x	1	1	0
1	0	x	0	x

000 → 0??
101 → 0??
110 → 0??

One x Becomes a 1 for S_1

For S_1, the x for state 101 became a 1, and the others became 0s.

S_1^+

S_2 \ S_1S_0	00	01	11	10
0	x	0	1	1
1	1	x	0	x

000 → 00?
101 → 01?
110 → 00?

One x Becomes a 1 for S_0

For S_0, the x for state 110 became a 1, and the others became 0s.

So what comes after 000 (black)?

Black again!

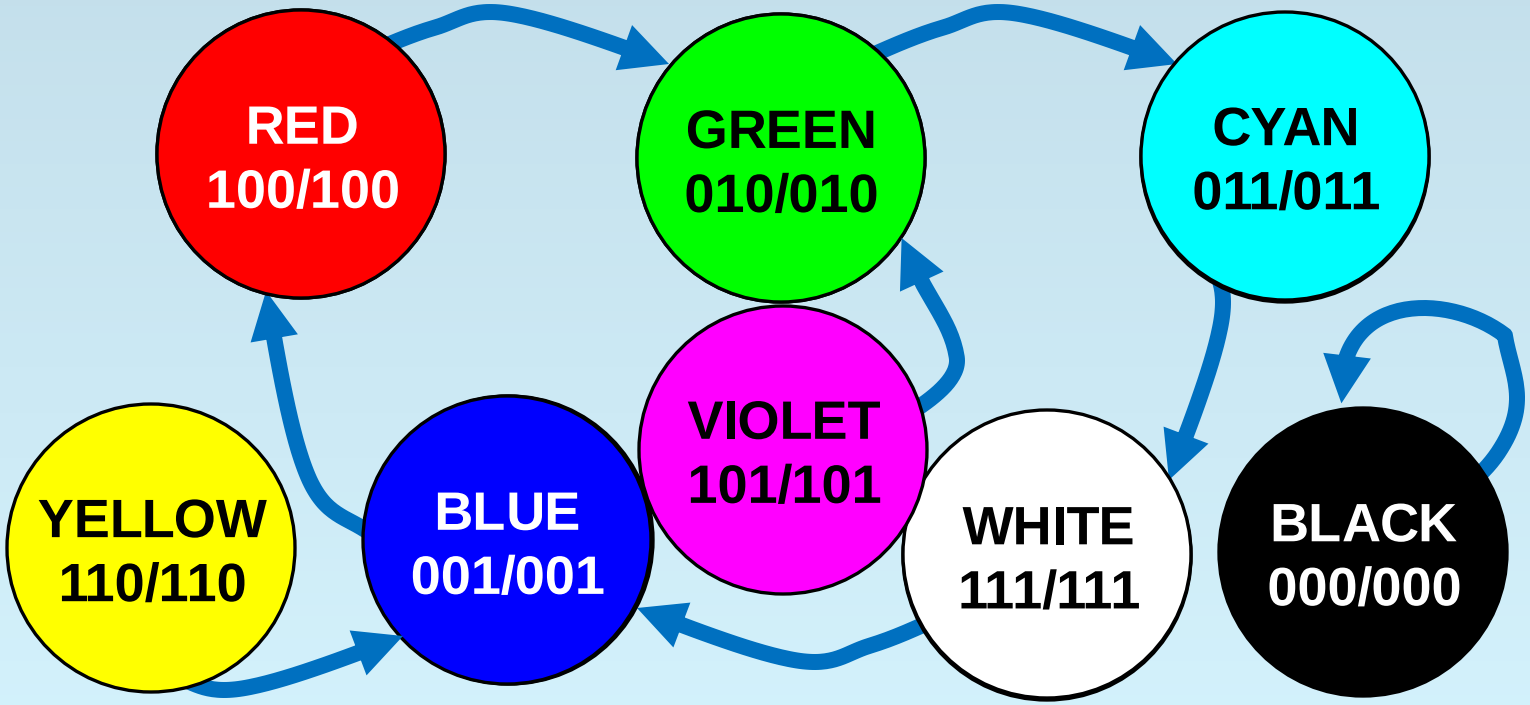
S_0^+

S_2 \ S_1S_0	00	01	11	10
0	x	0	1	1
1	0	x	1	x

000 → 000
101 → 010
110 → 001

Full Transition Diagram Illustrates Buggy Behavior

We can add these states to our diagram.



Avoid Bad States by Initializing the Counter State

What can we do? Let's add a way to initialize.

We can…

- **choose a specific (hardwired) initial state** at power-on (one from our loop*),
- **use muxes** to enable ourselves to set the state arbitrarily at any time,
- or use one signal to force the system into the loop, such as $\mathbf{S_0^+ = (S_1'INIT')'}$ (active low).

*Forcing all flip-flops to 0 doesn't help!

One Can Always Backtrack in the Design Process

Alternatively,

- we can go back to our K-maps and add loops.
- We may need to iterate a couple of times to find a design that always works.

We could also just choose specific next states for the states outside of our loop.

These approaches require more logic.