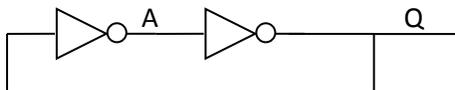


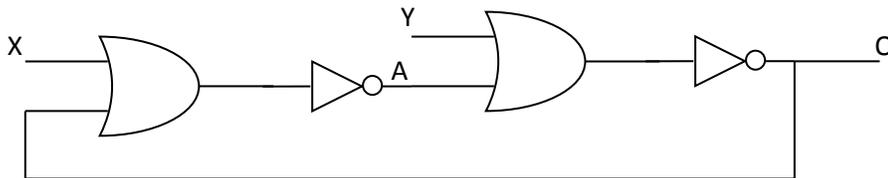
Sequential Logic (3.1 and 3.2. 3.2 is a long difficult section you really should read!)

One thing we have carefully avoided so far is feedback—all of our signals have gone from left-to-right. What if we don't do that?



Consider the figure on the left. It is a “bi-stable” device. That means there are two ways it could be stable. If $A=0$ then $Q=1$ and you'd expect that to hold its value (because A is then driven to 0 by Q). In the same way $A=1, Q=0$ is stable. So we've got two different sets of values for which this could be stable.

So is such a device useful? It would then *seem* like a good candidate for a memory device—after all it can keep one of two values. The problem is that we can't write to that device as both A and Q are already being driven by something else. So let's consider the following device instead



Now consider the above diagram. Recall that $0+X=X$. So if $X = Y = 0$, those OR gates act like wires. So *in that case* the device functions exactly the same as the pair of inverters.

Questions:

1. What is Q if $X=0$ and $Y=1$?
2. What is Q if $X=1$ and $Y=0$?
3. What is Q if $X=0$ and $Y=1$ and then X changes to a 0?

Definition

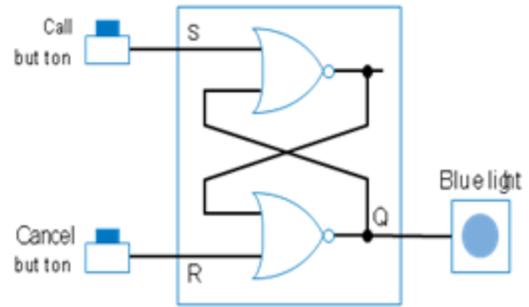
From our text:

A **sequential circuit** is a circuit whose output depends not only on the circuit's present inputs, but also on the circuit's present state, which is all the bits stored in the circuit. The circuit's state in turn depends on the past sequence of the circuit's input values.

The net effect is that we are using feedback (as shown above) to create **memory**. That memory allows us to have state (the information stored in the circuit).

S-R latch

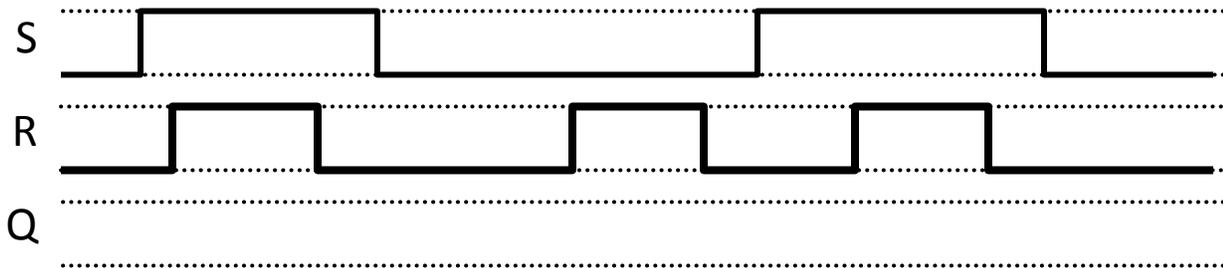
The above figure is redrawn on the right (figure from text), replacing the OR and NOT gates with NOR gates. And some wires moved around. This is the most common way to draw this device. It is called an SR latch.



Questions:

1. Fill in the truth table below
2. Complete the timing diagram.

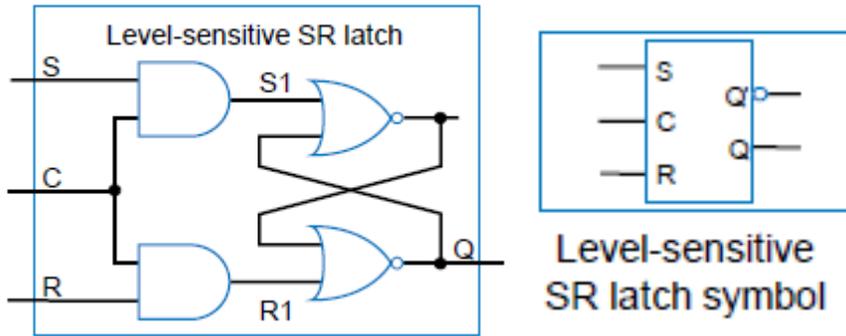
S	R	Q
0	0	
0	1	
1	0	
1	1	



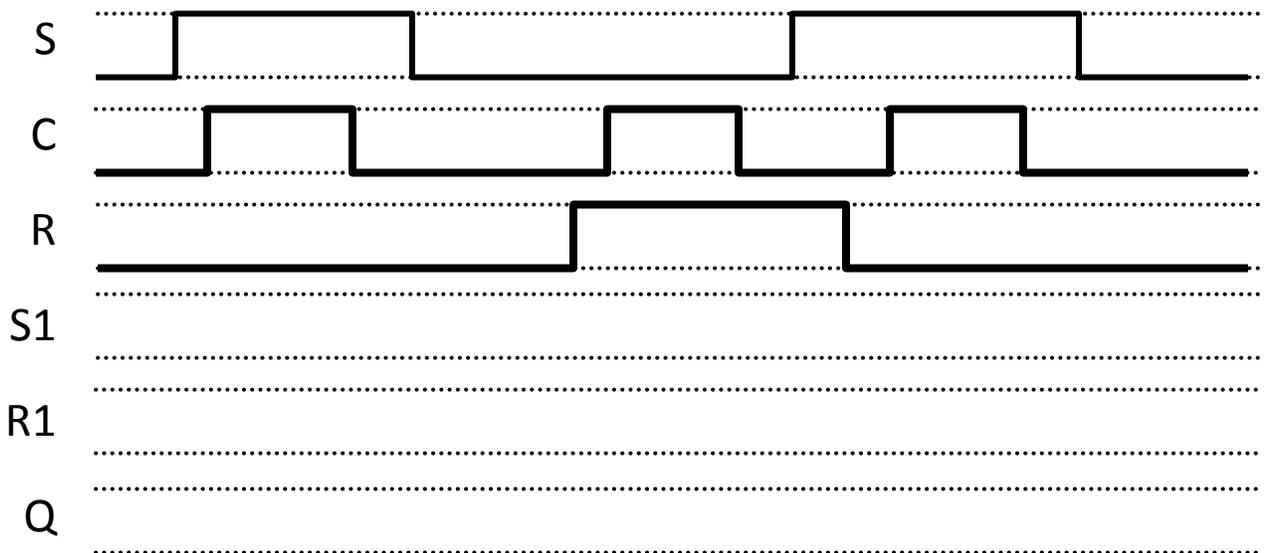
3. The big problem with an SR latch is what happens when the inputs go from $S=R=1$ to $S=R=0$. What happens in that case?

SR-latch with enable (which our text calls a level-sensitive SR latch)

One improvement: Add an “enable” or “clock”. Only allow S and R to change when C=1.

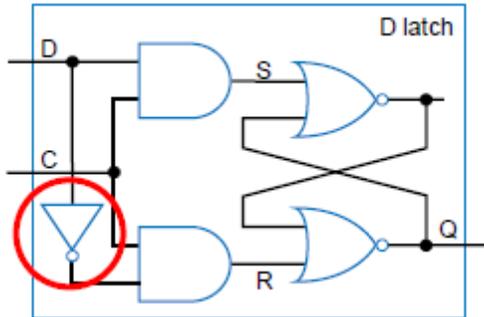


Fill in the timing diagram below.

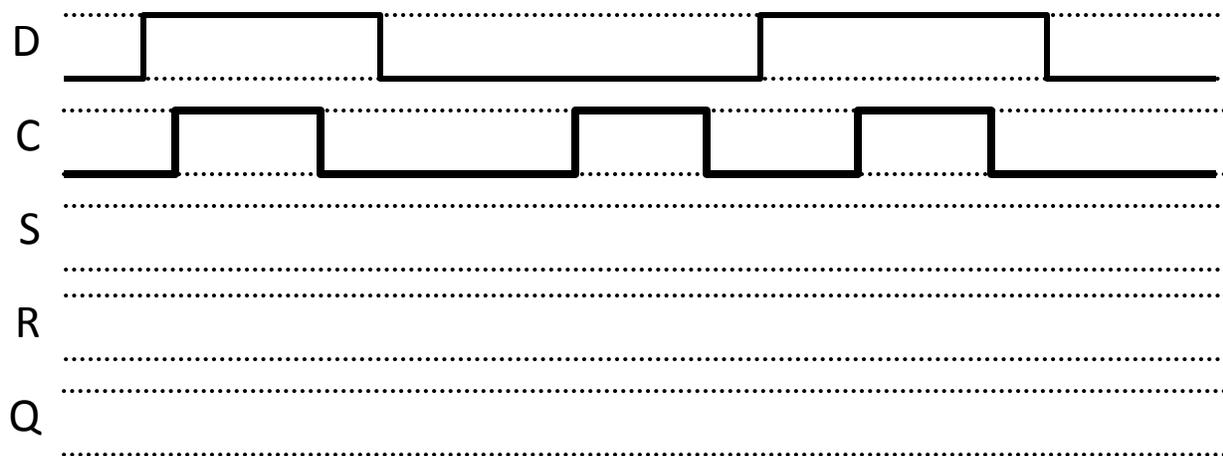


D latch

The D latch, shown below, is a very common type of latch. It has the property that S and R can't both be high at the same time.



Fill in the timing diagram below.



State machines (3.3)

Say I wish to have something occur over a time period.

- Maybe I want to send a message over a wire one bit at a time.
- Maybe I want to read a message coming over a wire (again one bit at a time) and do something based on that message.

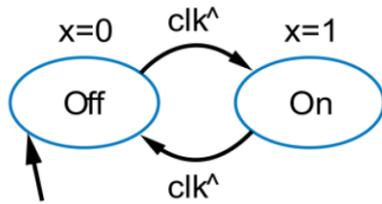
How do I do that?

Well, with combinational logic, I can't. There is no sense of time or history: inputs purely determine the outputs. So we need **sequential** logic—a device that can keep track of history and has a sense of timing.

Consider the following diagram from our text:

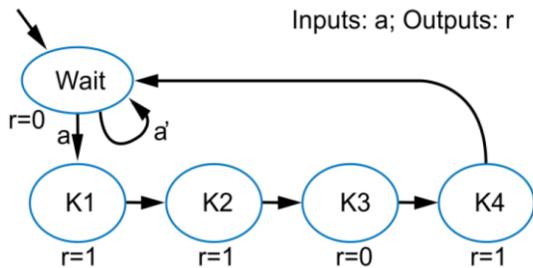
(The term clk^{\wedge} represents the rising edge of a clock)

Outputs: x



This “State Transition Diagram” describes a “Finite State Machine” All that it’s saying is that on every rising edge of some signal (clk) an output “ x ” will be changing from 1 to 0 or 0 to 1. The little arrow that’s hanging out by itself indicates the start state.

Our State Transition Diagram is a way to describe the desired behavior of a sequential circuit. Let’s steal another example from the book. Say we have a “smart key” for our car. When the key is put in the lock, the lock asks the key for its code. When asked for the code, the key provides the code “1101” one bit at a time.



This state transition diagram will generate the code “1101” one bit at a time when the signal “ a ” is set to 1.

Notice that the “ $clk^$ ” is missing. *It is always the case that a transition from one state to another will only occur on the rising edge of the clock.*

Your state transition diagrams should *never* include the clock. NEVER.

A comment on the text. Our author uses the term “state diagram” to describe what I’m calling a “state transition diagram”. Feel free to use either term, I will. However, he also sometimes doesn’t distinguish between the diagram and the underlying machine. That’s an important distinction.

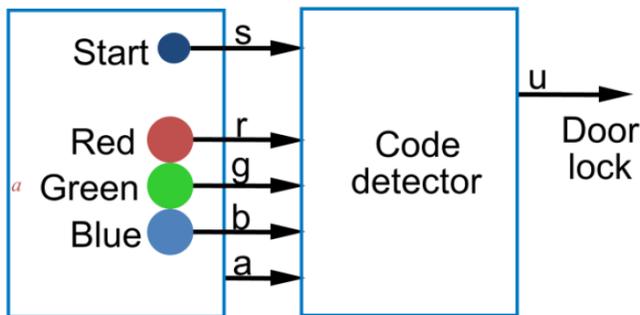
Let’s do some design.

Design a state diagram which has one input “ x ” and generates a “ z ” if (and only if) the last four values of x were “1100”.

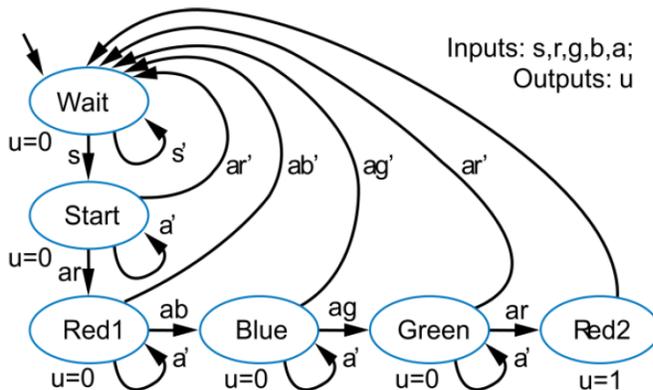
Another state transition diagram example (Chapter 3)

Design a state diagram which has one input “x” and generates a “z” if (and only if) the last four values of x were “1010”.

Let’s consider another problem from the book. Say we have a door with color-coded locks.



- Unlock door ($u=1$) only when buttons pressed in sequence:
 - start, then red, blue, green, red
- Input from each button: s, r, g, b
 - Also, output a indicates that some colored



Before we pick apart issues with this design, let’s notice some stuff.

1. From every state we always know exactly where to go.
 - a. Put differently, for every input in every state we have exactly one state to go to.
 - b. Notice that not all inputs need be used in each state...
2. We’ve made the initial state clear (how?)
3. Each state has all of the outputs defined (okay, only one in this case).

The above three things are all important properties of a state diagram!

Question

Why would this diagram not actually work in the real world? (Think about what would happen if you held the button for more than one clock tick (which you would)). How would you fix that?

The moral of the story here is that for embedded systems (things used by people) you need to keep in mind that we do have a clock even though it doesn’t show up in our state transition diagrams!