We'll describe the operation of the FSM for our combination lock using a state transition diagram.

Initially, the FSM has received no bits of the combination, a state we'll call SX.

In the state transition diagram, states are represented as circles, each labeled for now with a symbolic name chosen to remind us of what history it represents.

For this FSM, the unlock output U will be a function of the current state, so we'll indicate the value of U inside the circle.

Since in state SX we know nothing about past input bits, the lock should stay locked and so U = 0.

We'll indicate the initial state with a wide border on the circle.

We'll use the successive states to remember what we've seen so far of the input combination.

So if the FSM is in state SX and it receives a 0 input, it should transition to state S0 to remind us that we've seen the first bit of the combination of 0-1-1-0.

We use arrows to indicate transitions between states and each arrow has a label telling us when that transition should occur.

So this particular arrow is telling us that when the FSM is in state SX and the next input is a 0, the FSM should transition to state S0.

Transitions are triggered by the rising edge of the FSM's clock input.

Let's add the states for the remainder of the specified combination.

The rightmost state, S0110, represents the point at which the FSM has detected the specified sequence of inputs, so the unlock signal is 1 in this state.

Looking at the state transition diagram, we see that if the FSM starts in state SX, the input sequence 0-1-1-0 will leave the FSM in state S0110.

So far, so good.

What should the FSM do if an input bit is not the next bit in the combination?

For example, if the FSM is in state SX and the input bit is a 1, it still has not received any correct combination bits, so the next state is SX again.

Here are the appropriate non-combination transitions for the other states.

Note that an incorrect combination entry doesn't necessarily take the FSM to state SX.

For example, if the FSM is in state S0110, the last four input bits have been 0-1-1-0.

If the next input is a 1, then the last four inputs bits are now 1-1-0-1, which won't lead to an open lock.

But the last two bits might be the first two bits of a valid combination sequence and so the FSM transitions to S01, indicating that a sequence of 0-1 has been entered over the last two bits.

We've been working with an FSM where the outputs are function of the current state, called a Moore machine.

Here the outputs are written inside the state circle.

If the outputs are a function of both the current state and the current inputs, it's called a Mealy machine.

Since the transitions are also a function of the current state and current inputs, we'll label each transition with appropriate output values using a slash to separate input values from output values.

So, looking at the state transition diagram on the right, suppose the FSM is in state S3.

If the input is a 0, look for the arrow leaving S3 labeled "0/".

The value after the slash tells us the output value, in this case 1.

If the input had been a 1, the output value would be 0.

There are some simple rules we can use to check that a state transition diagram is well formed.

The transitions from a particular state must be mutually exclusive, i.e., for a each state, there can't be more than one transition with the same input label.

This makes sense: if the FSM is to operate consistently there can't be any ambiguity about the next state for a given current state and input.

By "consistently" we mean that the FSM should make the same transitions if it's restarted at the same starting state and given the same input sequences.

Moreover, the transitions leaving each state should be collectively exhaustive, i.e., there should a transition specified for each possible input value.

If we wish the FSM to stay in it's current state for that particular input value, we need to show a transition from the current state back to itself.

With these rules there will be exactly one transition selected for every combination of current state and input value.

All the information in a state transition diagram can be represented in tabular form as a truth table.

The rows of the truth table list all the possible combinations of current state and inputs.

And the output columns of the truth table tell us the next state and output value associated with each row.

If we substitute binary values for the symbolic state names, we end up with a truth table just like the ones we saw in Chapter 4.

If we have K states in our state transition diagram we'll need log_2(K) state bits, rounded up to the next integer since we don't have fractional bits!

In our example, we have a 5-state FSM, so we'll need 3 state bits.

We can assign the state encodings in any convenient way, e.g., 000 for the first state, 001 for the second state, and so on.

But the choice of state encodings can have a big effect on the logic needed to implement the truth table.

It's actually fun to figure out the state encoding that produces the simplest possible logic.

With a truth table in hand, we can use the techniques from Chapter 4 to design logic circuits that implement the combinational logic for the FSM.

Of course, we can take the easy way out and simply use a read-only memory to do the job!

In this circuit, a read-only memory is used to compute the next state and outputs from the current state and inputs.

We're encoding the 5 states of the FSM using a 3-bit binary value, so we have a 3-bit state register.

The rectangle with the edge-triggered input is schematic shorthand for a multi-bit register.

If a wire in the diagram represents a multi-bit signal, we use a little slash across the wire with a number to indicate how many bits are in the signal.

In this example, both current_state and next_state are 3-bit signals.

The read-only memory has a total of 4 input signals - 3 for the current state and 1 for the input value - so the read-only memory has 2^4 or 16 locations, which correspond to the 16 rows in the truth table.

Each location in the ROM supplies the output values for a particular row of the truth table.

Since we have 4 output signals - 3 for the next state and 1 for the output value - each location supplies 4 bits of information.

Memories are often annotated with their number of locations and the number of bits in each location.

So our memory is a 16-by-4 ROM: 16 locations of 4-bits each.

Of course, in order for the state registers to work correctly, we need to ensure that the dynamic discipline is obeyed.

We can use the timing analysis techniques described at the end of Chapter 5 to check that this is so.

For now, we'll assume that the timing of transitions on the inputs are properly synchronized with the rising edges of the clock.

So now we have the FSM abstraction to use when designing the functionality of a sequential logic system, and a general-purpose circuit implementation of the FSM using a ROM and a multi-bit state register.

Recapping our design choices: the output bits can be strictly a function of the current state (the FSM would then be called a Moore machine), or they can be a function of both the current state and current inputs, in which case the FSM is called a Mealy machine.

We can choose the number of state bits - S state bits will give us the ability to encode 2^S possible states.

Note that each extra state bit DOUBLES the number of locations in the ROM!

So when using ROMs to implement the necessary logic, we're very interested in minimizing the number of state bits.

The waveforms for our circuitry are pretty straightforward.

The rising edge of the clock triggers a transition in the state register outputs.

The ROM then does its thing, calculating the next state, which becomes valid at some point in the clock cycle.

This is the value that gets loaded into the state registers at the next rising clock edge.

This process repeats over-and-over as the FSM follows the state transitions dictated by the state transition diagram.

There are a few housekeeping details that need our attention.

On start-up we need some way to set the initial contents of the state register to the correct encoding for the initial state.

Many designs use a RESET signal that's set to 1 to force some initial state and then set to 0 to start execution.

We could adopt that approach here, using the RESET signal to select an initial value to be loaded into the state register.

In our example, we used a 3-bit state encoding which would allow us to implement an FSM with up to 2^3 = 8 states.

We're only using 5 of these encodings, which means there are locations in the ROM we'll never access.

If that's a concern, we can always use logic gates to implement the necessary combinational logic instead of ROMs.

Suppose the state register somehow got loaded with one of the unused encodings?

Well, that would be like being in a state that's not listed in our state transition diagram.

One way to defend against this problem is design the ROM contents so that unused states always point to the initial state.

In theory the problem should never arise, but with this fix at least it won't lead to unknown behavior.

We mentioned earlier the interesting problem of finding a state encoding that minimized the combinational logic.

There are computer-aided design tools to help do this as part of the larger problem of finding minimal logic implementations for Boolean functions.

Mr. Blue is showing us another approach to building the state register for the combination lock: use a shift register to capture the last four input bits, then simply look at the recorded history to determine if it matches the combination.

No fancy next0state logic here!

Finally, we still have to address the problem of ensuring that input transitions don't violate the dynamic discipline for the state register.

We'll get to this in the last section of this chapter.