MITOCW | MIT6_004S17_06-02-01_300k

In the last chapter we developed sequential logic, which contains both combinational logic and memory components.

The combinational logic cloud is an acyclic graph of components that obeys the static discipline.

The static discipline guarantees if we supply valid and stable digital inputs, then we will get valid and stable digital outputs by some specified interval after the last input transition.

There's also a functional specification that tells us the output values for every possible combination of input values.

In this diagram, there are k+m inputs and k+n outputs, so the truth table for the combinational logic will have 2^(k+m) rows and k+n output columns.

The job of the state registers is to remember the current state of the sequential logic.

The state is encoded as some number k of bits, which will allow us to represent 2^k unique states.

Recall that the state is used to capture, in some appropriate way, the relevant history of the input sequence.

To the extent that previous input values influence the operation of the sequential logic, that happens through the stored state bits.

Typically the LOAD input of the state registers is triggered by the rising edge of a periodic signal, which updates the stored state with the new state calculated by the combinational logic.

As designers we have several tasks:

first we must decide what output sequences need to be generated in response to the expected input sequences.

A particular input may, in fact, generate a long sequence of output values.

Or the output may remain unchanged while the input sequence is processed, step-by-step, where the FSM is remembering the relevant information by updating its internal state.

Then we have to develop the functional specification for the logic so it calculates the correct output and next state values.

Finally, we need to come up with an actual circuit diagram for sequential logic system.

All the tasks are pretty interesting, so let's get started!

As an example sequential system, let's make a combination lock.

The lock has a 1-bit input signal, where the user enters the combination as a sequence of bits.

There's one output signal, UNLOCK, which is 1 if and only if the correct combination has been entered.

In this example, we want to assert UNLOCK, i.e., set UNLOCK to 1, when the last four input values are the sequence 0-1-1-0.

Mr. Blue is asking a good question: how many state bits do we need?

Do we have to remember the last four input bits?

In which case, we'd need four state bits.

Or can we remember less information and still do our job?

Aha! We don't need the complete history of the last four inputs, we only need to know if the most recent entries represent some part of a partially-entered correct combination.

In other words if the input sequence doesn't represent a correct combination, we don't need to keep track of exactly how it's incorrect, we only need to know that is incorrect.

With that observation in mind, let's figure out how to represent the desired behavior of our digital system.

We can characterize the behavior of a sequential system using a new abstraction called a finite state machine, or FSM for short.

The goal of the FSM abstraction is to describe the input/output behavior of the sequential logic, independent of its actual implementation.

A finite state machine has a periodic CLOCK input.

A rising clock edge will trigger the transition from the current state to the next state.

The FSM has a some fixed number of states, with a particular state designated as the initial or starting state when the FSM is first turned on.

One of the interesting challenges in designing an FSM is to determine the required number of states since there's often a tradeoff between the number of state bits and the complexity of the internal combinational logic required to compute the next state and outputs.

There are some number of inputs, used to convey all the external information necessary for the FSM to do its job.

Again, there are interesting design tradeoffs.

Suppose the FSM required 100 bits of input.

Should we have 100 inputs and deliver the information all at once?

Or should we have a single input and deliver the information as a 100-cycle sequence?

In many real world situations where the sequential logic is *much* faster than whatever physical process we're trying to control,

we'll often see the use of bit-serial inputs where the information arrives as a sequence, one bit at a time.

That will allow us to use much less signaling hardware, at the cost of the time required to transmit the information sequentially.

The FSM has some number outputs to convey the results of the sequential logic's computations.

The comment before about serial vs. parallel inputs applies equally to choosing how information should be encoded on the outputs.

There are a set of transition rules, specifying how the next state S-prime is determined from the current state S and the inputs I.

The specification must be complete, enumerating S-prime for every possible combination of S and I.

And, finally, there's the specification for how the output values should be determined.

The FSM design is often a bit simpler if the outputs are strictly a function of the current state S, but, in general, the outputs can be a function of both S and the current inputs.

Now that we have our abstraction in place, let's see how to use it to design our combinational lock.