# MITOCW | MIT6_004S17_04-02-01_300k

In this lecture you'll learn various techniques for creating combinational logic circuits that implement a particular functional specification.

A functional specification is part of the static discipline we use to create the combinational logic abstraction of a circuit.

One approach is to use natural language to describe the operation of a device.

This approach has its pros and cons.

In its favor, natural language can convey complicated concepts in surprisingly compact form and it is a notation that most of us know how to read and understand.

But, unless the words are very carefully crafted, there may be ambiguities introduced by words with multiple interpretations or by lack of completeness since it's not always obvious whether all eventualities have been dealt with.

There are good alternatives that address the shortcomings mentioned above.

Truth tables are a straightforward tabular representation that specifies the values of the outputs for each possible combination of the digital inputs.

If a device has N digital inputs, its truth table will have 2^N rows.

In the example shown here, the device has 3 inputs, each of which can have the value 0 or the value 1.

There are 2*2*2 = 2^3 = 8 combinations of the three input values, so there are 8 rows in the truth table.

It's straightforward to systematically enumerate the 8 combinations, which makes it easy to ensure that no combination is omitted when building the specification.

And since the output values are specified explicitly, there isn't much room for misinterpreting the desired functionality!

Truth tables are an excellent choice for devices with small numbers of inputs and outputs.

Sadly, they aren't really practical when the devices have many inputs.

If, for example, we were describing the functionality of a circuit to add two 32-bit numbers, there would be 64 inputs altogether and the truth table would need 2^64 rows.

Hmm, not sure how practical that is!

If we entered the correct output value for a row once per second, it would take 584 billion years to fill in the table!

Another alternative specification is to use Boolean equations to describe how to compute the output values from the input values using Boolean algebra.

The operations we use are the logical operations AND, OR, and XOR, each of which takes two Boolean operands, and NOT which takes a single Boolean operand.

Using the truth tables that describe these logical operations, it's straightforward to compute an output value from a particular combination of input values using the sequence of operations laid out in the equation.

Let me say a quick word about the notation used for Boolean equations.

Input values are represented by the name of the input, in this example one of "A", "B", or "C".

The digital input value 0 is equivalent to the Boolean value FALSE and the digital input value 1 is equivalent to the Boolean value TRUE.

The Boolean operation NOT is indicated by a horizontal line drawn above a Boolean expression.

In this example, the first symbol following the equal sign is a "C" with line above it, indicating that the value of C should be inverted before it's used in evaluating the rest of the expression.

The Boolean operation AND is represented by the multiplication operation using standard mathematical notation.

Sometimes we'll use an explicit multiplication operator - usually written as a dot between two Boolean expressions - as shown in the first term of the example equation.

Sometimes the AND operator is implicit as shown in the remaining three terms of the example equation.

The Boolean operation OR is represented by the addition operation, always shown as a "+" sign.

Boolean equations are useful when the device has many inputs.

And, as we'll see, it's easy to convert a Boolean equation into a circuit schematic.

Truth tables and Boolean equations are interchangeable.

If we have a Boolean equation for each output, we can fill in the output columns for a row of the truth table by evaluating the Boolean equations using the particular combination of input values for that row.

For example, to determine the value for Y in the first row of the truth table, we'd substitute the Boolean value FALSE for the symbols A, B, and C in the equation and then use Boolean algebra to compute the result.

We can go the other way too.

We can always convert a truth table into a particular form of Boolean equation called a sum-of-products.

Let's see how… Start by looking at the truth table and answering the question "When does Y have the value 1?" Or, in the language of Boolean algebra: "When is Y TRUE?".

Well, Y is TRUE when the inputs correspond to row 2 of the truth table, OR to row 4, OR to rows 7 OR 8.

Altogether there are 4 combinations of inputs for which Y is TRUE.

The corresponding Boolean equation thus is the OR for four terms, where each term is a boolean expression which evaluates to TRUE for a particular combination of inputs.

Row 2 of the truth table corresponds to C=0, B=0, and A=1.

The corresponding Boolean expression is (NOT C) AND (NOT B) AND A, an expression that evaluates to TRUE if and only if C is 0, B is 0, and A is 1.

The boolean expression corresponding to row 4 is (NOT C) AND B AND A.

And so on for rows 7 and 8.

The approach will always give us an expression in the form of a sum-of-products.

"Sum" refers to the OR operations and "products" refers to the groups of AND operations.

In this example we have the sum of four product terms.

Our next step is to use the Boolean expression as a recipe for constructing a circuit implementation using combinational logic gates.

As circuit designers we'll be working with a library of combinational logic gates, which either is given to us by the integrated circuit manufacturer, or which we've designed ourselves as CMOS gates using NFET and PFET switches.

One of the simplest gates is the inverter, which has the schematic symbol shown here.

The small circle on the output wire indicates an inversion, a common convention used in schematics.

We can see from its truth table that the inverter implements the Boolean NOT function.

The AND gate outputs 1 if and only if the A input is 1 *and* the B input is 1, hence the name "AND".

The library will usually include AND gates with 3 inputs, 4 inputs, etc., which produce a 1 output if and only if all of their inputs are 1.

The OR gate outputs 1 if the A input is 1 *or* if the B input is 1, hence the name "OR".

Again, the library will usually include OR gates with 3 inputs, 4 inputs, etc., which produce a 1 output when at least one of their inputs is 1.

These are the standard schematic symbols for AND and OR gates.

Note that the AND symbol is straight on the input side, while the OR symbol is curved.

With a little practice, you'll find it easy to remember which schematic symbols are which.

Now let's use these building blocks to build a circuit that implements a sum-of-products equation.

The structure of the circuit exactly follows the structure of the Boolean equation.

We use inverters to perform the necessary Boolean NOT operations.

In a sum-of-products equation the inverters are operating on particular input values, in this case A, B and C. To keep the schematic easy to read we've used a separate inverter for each of the four NOT operations in the Boolean equation, but in real life we might invert the C input once to produce a NOT-C signal, then use that signal whenever a NOT-C value is needed.

Each of the four product terms is built using a 3-input AND gate.

And the product terms are ORed together using a 4-input OR gate.

The final circuit has a layer of inverters, a layer of AND gates and final OR gate.

In the next section we'll talk about how to build AND or OR gates with many inputs from library components with fewer inputs.

The propagation delay for a sum-of-products circuit looks pretty short: the longest path from inputs to outputs includes an inverter, an AND gate and an OR gate.

Can we really implement any Boolean equation in a circuit with a tPD of three gate delays?

Actually not, since building ANDs and ORs with many inputs will require additional layers of components, which will increase the propagation delay.

We'll learn about this in the next section.

The good news is that we now have straightforward techniques for converting a truth table to its corresponding sum-of-products Boolean equation, and for building a circuit that implements that equation.