ENEL260 CA11: More Boolean Algebra

Ciaran Moore

"... logic is the mathematics of the two Boolean values TRUE and FALSE"
– Leslie Lamport

Designing the next-state and output logic for FSMs requires using truth tables and Boolean algebra to find a combinational logic design. This essentially brings us full-circle, since we began this part of the course by looking at Boolean algebra. In this final lecture we'll look at Boolean algebra again, and consider some visual techniques to help with simplifying Boolean functions.

In the first Computer Architecture lecture we looked at Boolean functions of logical variables. We learned that a Boolean function is a rule for transforming the values of one or more logical variables (the "inputs") into another logical value (the "output").

Boolean Functions

We can precisely define the "rule" represented by a Boolean function using a *truth table*. The truth table for a function lists *every possible combination* of input values that the function can be applied to, and shows what the output value will be for each of those input combinations.

Example: If we want to define the logical function $AND(a, b)$, which will output a 1 only when *both* of the inputs are 1, we can draw up a truth table that shows all of the possible value a and b might take on and what the resulting output will be for each combination:

Inputs		Output
a	b	$AND(a, b)$
0	0	0
0	1	0
1	0	0
1	1	1

You can think of the truth table as being like a *dictionary* or *lookup table* for finding an output value based on the input values. In fact, if we were trying to implement the $AND(a, b)$ Boolean function in Python, we might use a `dict` data structure:

You could, of course, also just implement the function as returning `a == 1 and b == 1`, since Python has a built-in and operator.

```python
def AND(a, b):
    truth_table = {(0, 0): 0,
                   (0, 1): 0,
                   (1, 0): 0,
                   (1, 1): 1}
    return truth_table[(a, b)]
```

Example: The logical function $XOR(a, b)$ outputs a 1 when *one* (but not both) of the inputs is 1:

$XOR(a, b)$ is the "exclusive or" function.

Inputs		Output
a	b	$XOR(a, b)$
0	0	0
0	1	1
1	0	1
1	1	0

In Python, that becomes:

```
def XOR(a, b):
    truth_table = {(0, 0): 0,
                   (0, 1): 1,
                   (1, 0): 1,
                   (1, 1): 0}
    return truth_table[(a, b)]
```

In addition to using a truth table to define a Boolean function, we can also write the function as a *Boolean expression* that uses the basic Boolean operators (AND, OR, and NOT functions) to specify the relationship between input values and output values:

- $AND(a, b) = 1$ when *both* a and b are 1 (usually written as $a \cdot b$, and abbreviated to $a\,b$)
- $OR(a, b) = 1$ when *at least* one of a or b is 1 (usually written as $a + b$)
- $NOT(a) = 1$ when a is 0 (usually written as $\overline{a}$)

Example: The logical function $XOR(a, b)$ can be defined using the AND, OR, and NOT Boolean functions as the expression

It should be fairly obvious how this expression-based definition would translate into Python.

$$XOR(a, b) = OR(AND(NOT(a), b), AND(a, NOT(b)),$$

or, using a more compact notation,

$$XOR(a, b) = \overline{a}b + a\overline{b}.$$

To confirm that this expression is correct, we can form a truth table showing the terms of the expression

a	b	$\overline{a}b$	$a\overline{b}$	$\overline{a}b + a\overline{b}$
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

The final column of the truth table is identical for the truth table for $XOR(a, b)$ in the example above.

From Truth Table to Expression

Since truth tables and Boolean expressions are both ways of defining a Boolean function, it should be possible to go back and forth between them. We can form a truth table from a Boolean expression by simply evaluating the expression for every possible combination of input values. But what about finding an expression from a truth table?

The simplest case is a truth table for which the output only has a single row that contains a 1, since we know the the Boolean expression we come up with must produce a 1 for only one possible combination of input values.

Example: The truth table

a	b	c	$f(a,b,c)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

defines a Boolean function that produces a 1 **only** when $a = 1$ AND $b = 0$ (i.e., when $\overline{b} = 1$) AND $c = 1$. In other words,

$$f(a,b,c) = a\overline{b}c$$

Now consider an arbitrary function of three variables, $z = f(a,b,c)$, with z given by the truth table:

a	b	c	z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

We can think of this as a combination of four separate functions that each produce a 1 on a single row. We already know how to find the Boolean expression for a function that will produce a 1 on any single row (define a function that ANDs together every input or its inverse, as above). So we can form a Boolean expression for z by ORing together the combinations of the inputs associated with each of the rows which generate a '1' at the output. Thus by inspection,

$$z = \bar{a}\,\bar{b}\,c + a\,\bar{b}\,\bar{c} + a\,\bar{b}\,c + a\,b\,c \qquad (1)$$

This is known as the "sum-of-products" (or "sum-of-minterms") form for z. Note that each *minterm* on the RHS contains all 3 input variables.

A *minterm* is an AND (or "product") term that contains all of the input variables to a Boolean function, with each variable appearing *exactly once* (in either positive or inverted form).

Simplifying Expressions

WE CAN IMPLEMENT a combinational logic circuit for z which uses 4×3-input *AND* gates and a single 4-input *OR* gate as represented by Eq. 1. However, that is likely to be somewhat wasteful of electronics, as we'll now see.

In the second truth table below, three terms of two variables are specified in the middle section. Work out what belongs in the RH column (i.e., take the *OR* of the middle columns).

a	b	c	$a\,\bar{b}$	$a\,c$	$\bar{b}\,c$	$a\,\bar{b} + a\,c + \bar{b}\,c$
0	0	0	0	0	0	
0	0	1	0	0	1	
0	1	0	0	0	0	
0	1	1	0	0	0	
1	0	0	1	0	0	
1	0	1	1	1	1	
1	1	0	0	0	0	
1	1	1	0	1	0	

You should be able to demonstrate that the RH column you have filled in is identical to that for z in the original truth table. That being the case,

$$z = \bar{a}\,\bar{b}\,c + a\,\bar{b}\,\bar{c} + a\,\bar{b}\,c + a\,b\,c = a\,\bar{b} + a\,c + \bar{b}\,c\,. \qquad (2)$$

Thus only 3×2-input *AND* gates and a single 3-input *OR* gate are needed to generate exactly the same logical result.

How can we derive this result in some systematic way? In Lecture 1 we saw the rules of Boolean algebra:

See https://en.wikipedia.org/wiki/Boolean_algebra

The rules of Boolean algebra can be used to manipulate Boolean expressions to find simpler versions, in the same way that the rules of classical algebra let you simplify numercal expressions.

Identity	$a \cdot 1 = a$ and $a + 0 = a$
Annihilation	$a \cdot 0 = 0$ and $a + 1 = 1$
Or-Complement	$a + \bar{a} = 1$
And-Complement	$a \cdot \bar{a} = 0$
Associative	$a + (b + c) = (a + b) + c, \quad a(bc) = (ab)c$
Commutative	$a + b = b + a, \quad ab = ba$
Distributive	$a(b + c) = ab + ac, \quad a + (bc) = (a + b) \cdot (a + c)$
Double Negation	$\bar{\bar{a}} = a$
Absorption	$a + (a \cdot b) = a$ and $a \cdot (a + b) = a$
Idempotence	$a + a = a$ and $a \cdot a = a$
De Morgan's Laws	$\overline{a + b} = \bar{a} \cdot \bar{b}$ $\overline{a \cdot b} = \bar{a} + \bar{b}$

One approach to simplifying our example is to consider pairings of the terms on the LHS, and apply the Distributive, Or-Complement, and Identity rules:

$$\begin{aligned} a\,\overline{b}\,\overline{c} + a\,\overline{b}\,c &= a\,\overline{b}(\overline{c}+c) = a\,\overline{b} \\ a\,\overline{b}\,c + a\,b\,c &= a(\overline{b}+b)c = a\,c \\ \overline{a}\,\overline{b}\,c + a\,\overline{b}\,c &= (\overline{a}+a)\overline{b}\,c = \overline{b}\,c \end{aligned}$$

Importantly, note that one of the terms from the RHS of Eq. (2) appears more than once on the LHS of the set of equations above. This illustrates a difference between logical relationships and arithmetic relationships: for logical variables, $xyz + xyz = xyz$ (Idempotence).

Example: To simplify $y = \overline{a}\,\overline{b}\,c + a\,\overline{b}\,\overline{c} + a\,\overline{b}\,c + a\,b\,\overline{c}$ we can apply several rules of Boolean algebra:

$$\begin{aligned} y &= \overline{a}\,\overline{b}\,c + a\,\overline{b}\,\overline{c} + a\,\overline{b}\,c + a\,b\,\overline{c} \\ &= \overline{a}\,\overline{b}\,c + a\,\overline{b}\,c + a\,\overline{c}\,\overline{b} + a\,\overline{c}\,b \quad [\mathit{Commutative}] \\ &= (\overline{a}+a)\overline{b}\,c + a\,\overline{c}(\overline{b}+b) \quad [\mathit{Distributive}] \\ &= 1\cdot\overline{b}\,c + a\,\overline{c}\cdot 1 \quad [\mathit{OrComplement}] \\ &= \overline{b}\,c + a\,\overline{c} \quad [\mathit{Identity}] \end{aligned}$$

Example: To simplify $x = \overline{a}\,\overline{b}\,c + \overline{a}\,b\,c + a\,\overline{b}\,c + a\,\overline{b}\,\overline{c}$ we can apply the following rules of Boolean algebra:

$$\begin{aligned} x &= \overline{a}\,\overline{b}\,c + \overline{a}\,b\,c + a\,\overline{b}\,c + a\,\overline{b}\,\overline{c} \\ &= \overline{a}\,c\,\overline{b} + \overline{a}\,c\,b + a\,\overline{b}\,c + a\,\overline{b}\,\overline{c} \quad [\mathit{Commutative}] \\ &= \overline{a}\,c(\overline{b}+b) + a\,\overline{b}(c+\overline{c}) \quad [\mathit{Distributive}] \\ &= \overline{a}\,c + a\,\overline{b} \quad [\mathit{OrComplement}\ \&\ \mathit{Identity}] \end{aligned}$$

Example: To simplify $p = \overline{x}\,y\,\overline{z} + x\,\overline{y}\,\overline{z} + x\,\overline{y}\,z + x\,y\,\overline{z} + x\,y\,z$ we can apply the following rules of Boolean algebra:

$$\begin{aligned} p &= \overline{x}\,y\,\overline{z} + x\,\overline{y}\,\overline{z} + x\,\overline{y}\,z + x\,y\,\overline{z} + x\,y\,z \\ &= \overline{x}\,y\,\overline{z} + x(\overline{y}\,\overline{z} + \overline{y}\,z + y\,\overline{z} + y\,z) \quad [\mathit{Distributive}] \\ &= \overline{x}\,y\,\overline{z} + x \quad [\mathit{OrComplement}\ \&\ \mathit{Identity}] \end{aligned}$$

Example: To simplify the combinational logic circuit shown in Fig. 1 we want to:

(i) Find a Boolean expression for r.

(ii) Simplify that expression.

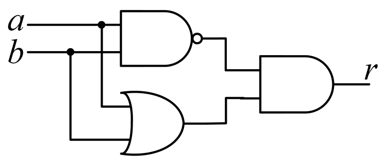


Figure 1: A logic circuit.

Looking at the circuit in Fig. 1 we can see that r is generated by a 2-input AND gate, i.e., using X and Y to represent the inputs to the AND gate:

$$r(a,b) = AND(X,Y) = X \cdot Y.$$

We can now consider X and Y, which are generated from a and b via two different gates. From Fig. 1, X is the output of a NAND gate, and Y is the output of an OR gate. Thus

$$\begin{aligned} X &= NAND(a,b) = \overline{ab} \\ Y &= OR(a,b) = a + b \end{aligned}$$

Substituting the expressions for X and Y into the expression for r, we get

$$r(a,b) = \overline{ab} \cdot (a+b)$$

We can now apply the rules of Boolean algebra to simplify the expression for $r(a,b)$:

$$\begin{aligned} r(a,b) &= \overline{ab} \cdot (a+b) \\ &= (\bar{a} + \bar{b}) \cdot (a+b) \quad [De\ Morgan] \\ &= \bar{a}a + \bar{b}a + \bar{a}b + \bar{b}b \quad [Distributive] \\ &= a\bar{b} + \bar{a}b \quad [AndComplement] \end{aligned}$$

This is equivalent to the expression for the standard XOR gate.

"Don't Care" Inputs

In some truth tables we know (or can easily see) that there are situations where the output is *the same* regardless of the value a particular input takes on. In those situations, we can make the truth table more compact by marking those input values as "don't cares" (X instead of 0 or 1), and eliminating any duplicate rows. This is essentially the same as performing an algebraic simplification.

Example: The Boolean function defined by the truth table

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

will produce a 1 whenever $a = 1$, regardless of what values b and c take on. The truth table can thus be made more compact by placing an X in the b and c columns for the row(s) in which $a = 1$:

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	X	X	1

The corresponding Boolean expression is

$$y = a + \bar{a}(\bar{b}\bar{c} + bc).$$

The use of "don't care" input values here is equivalent to noting that the sum-of-minterms $a\bar{b}\bar{c} + ab\bar{c} + a\bar{b}c + abc$ can be simplfied to $a(\bar{b}\bar{c} + b\bar{c} + \bar{b}c + bc) = a$.

The Karnaugh Map

[Not examinable]

The principles behind the Karnaugh Map are also used by modern logic optimization software, so it's useful to understand how K-Maps work.

THE *Karnaugh Map* IS A VISUAL METHOD for simplifying Boolean expressions. Each of the cells in the K-Map corresponds to one of the minterms that make up a sum-of-products. The K-Map is designed to place minterms that differ by only a single variable next to each other. Groupings of adjacent[1] cells that all contain 1 represent opportunities to combine terms and eliminate variables.

[1] "Adjacent" horizontally or vertically, but not diagonally. Note that the K-Map "wraps-around" at the edges, so the top row is adjacent to the bottom, and the rightmost column is adjacent to the leftmost.
K-Maps for functions with greater than 4 inputs have been proposed, but are not all that practical to use.

A K-Map is a tabular representation of a 3-input or 4-input logic function built by using 4 columns to represent all of the possible combinations of two of the inputs, and either 2 or 4 rows to represent all of the possible combinations of the other inputs. Thus, for a 4-input logic function $f(a, b, c, d)$ the K-Map table will look like the following:

Note the ordering of the columns (and rows), which ensures that only one variable changes between adjacent columns (e.g., $\bar{a}b$ is adjacent to ab). This ordering is known as a Gray code.

	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$
$\bar{c}\bar{d}$				
$\bar{c}d$				
cd				
$c\bar{d}$				

$f(a, b, c, d)$

To use a K-Map for simplifying a Boolean function we need to:

1. Find a sum-of-minterms expression for the function (or build a truth table).
2. Construct a K-Map table that has a 1 in every cell that corresponds to a minterm.

3. Find the largest possible groupings of adjacent 1s. The groupings must be one, two, four, eight, or sixteen cells in size[2], and can overlap.

4. Write a simplified expression for the function by ORing together the terms represented by each grouping.

[2] Sizes like three or six represent invalid simplifications, since they don't contain all possible combinations of whatever variable we're attempting to eliminate.

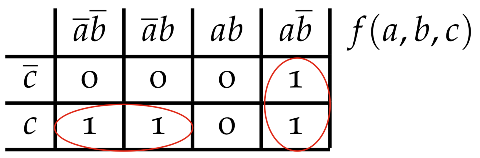
Example: In one of the earlier examples, the following expression is given for simplification:

$x = f(a,b,c) \;=\; \bar{a}\bar{b}c + \bar{a}bc + a\bar{b}c + a\bar{b}\bar{c}$

Let's find a Karnaugh Map simplification of this function. The function is already in sum-of-minterms form, which makes constructing the K-Map straightforward:

	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$	$f(a,b,c)$
$\bar{c}$	0	0	0	1	
c	1	1	0	1	

We can find two groupings of 1s in the K-Map, as shown in Fig. 2. The lower-left group in Fig. 2 includes both a column for $\bar{b}$ and a column for b, thus eliminating b as a variable and allowing the group to be represented by the term $\bar{a}c$. Similarly, the righthand group eliminates c, since it covers both the c and $\bar{c}$ rows.



	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$	$f(a,b,c)$
$\bar{c}$	0	0	0	1	
c	1	1	0	1	

Figure 2: K-Map with groups of adjacent 1s circled.

The simplified expression for $f(a,b,c)$ can be found by ORing the expressions representing each group:

$$f(a,b,c) = \bar{a}c + a\bar{b}$$

Example: Earlier we looked at an example of a truth table with "don't care" inputs:

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	X	X	1

How does this get turned into a K-Map? Where $a = 0$ (i.e., $\bar{a}$) we have two minterms that can be treated as usual. Where $a = 1$, we know that *every* cell associated with a (the shaded cells in the table below) should be marked with a 1:

	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$	y
$\bar{c}$	1	0	1	1	
c	0	1	1	1	

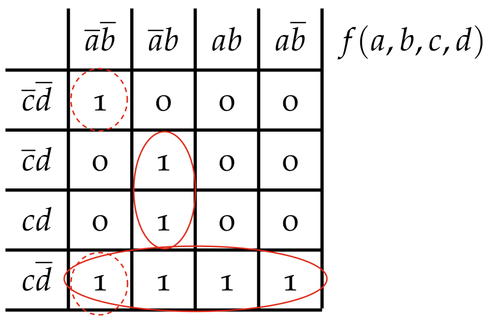
Example: A 4-variable logic function is defined in the truth table below.

a	b	c	d	t
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

The sum-of-minterms expression for this truth table is:

$$t = f(a,b,c,d) = \bar{a}\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + \bar{a}b\bar{c}d + \bar{a}bc\bar{d} + \bar{a}bcd + a\bar{b}c\bar{d} + abc\bar{d}$$

Based on the sum-of-minterms expression, we can build the Karnaugh map in Fig. 3.



	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$	$f(a,b,c,d)$
$\bar{c}\bar{d}$	1	0	0	0	
$\bar{c}d$	0	1	0	0	
cd	0	1	0	0	
$c\bar{d}$	1	1	1	1	

Figure 3: K-Map with groups of adjacent 1s circled.

In Fig. 3 the groups are:

- $c\bar{d}$ (the large group; completely eliminates a and b)
- $\bar{a}bd$ (eliminates c)
- $\bar{a}\bar{b}\,\bar{d}$ (takes advantage of the top and bottom rows being adjacent, and also partially overlaps with the first group)

The simplified expression for $f(a,b,c,d)$ can be found by ORing the expressions representing each group:

$$f(a,b,c,d) = c\bar{d} + \bar{a}bd + \bar{a}\bar{b}\,\bar{d}$$

"Don't Care" Outputs

In some situations we may not care what the output is for a particular combination of inputs. This most commonly happens when we know that certain combinations of inputs aren't possible. For example:

This is sometimes referred to as a "don't care input combination", which is not the same thing as what we've been calling a "don't care input".

- If we're designing a circuit to process information from a sliding switch that can only ever be in one of five different positions then we might know that we can't possibly receive any inputs outside the set $\{001, 010, 011, 100, 101\}$.
- If we're designing next-state logic for an FSM that we know doesn't use all of the state codes made possible by the number of bits used to encode the states (e.g. using 3 bits to encode 5 states leaves 3 unused codes).

If we mark the outputs associated with impossible input combinations as "don't cares" when we build a K-Map, we can give ourselves the flexibility to choose to treat the X's as either a 0 or a 1 depending on which will best help in simplifying the final Boolean expression.

***Caution:* while using "don't cares" can help with simplifying logic expressions, you need to ask yourself if you're *really sure* that a particular input combination can't happen. Or if you *really don't care* if the combination does happen as the result of some fault in the circuit.**

Example: Here's the K-Map for a 3-input logic circuit that should output a 1 when the input is 010 or 100:

	$\overline{a}\overline{b}$	$\overline{a}b$	ab	$a\overline{b}$	y
$\overline{c}$	0	1	0	1	
c	0	0	0	0	

This offers little opportunity for simplification:

$$y = \overline{a}b\overline{c} + a\overline{b}\overline{c}.$$

But if we know that the input *must* come from the set $\{001, 010, 011, 100, 101\}$, then we "don't care" what output is generated for inputs 000, 110, and 111:

	$\overline{a}\overline{b}$	$\overline{a}b$	ab	$a\overline{b}$	y
$\overline{c}$	X	1	X	1	
c	0	0	X	0	

With this K-Map, we could choose to treat the "don't cares" in the top row as 1, and the "don't care" in the bottom row as 0, resulting in the simplified expression:

$$y = \overline{c}.$$

This will have exactly the same behaviour as the previous expression for every *valid* input combination.

Summary

- Boolean functions can be defined as truth tables (specifying the output for all posible input combinations) or as Boolean expressions (specifying the output by using AND, OR, and NOT to combine input variables)
- Boolean algebra offers a way to simplify complex logic expressions, and to thereby reduce the compelxity of hardware designs
- Karnaugh Maps are a visual method for simplifying Boolean expressions