 ©2012 Steven S. Lumetta. All rights reserved.

ECE199JL: Introduction to Computer Engineering **Fall 2012**
Notes Set 2.2

Boolean Properties and Don't Care Simplification

This set of notes begins with a brief illustration of a few properties of Boolean logic, which may be of use to you in manipulating algebraic expressions and in identifying equivalent logic functions without resorting to truth tables. We then discuss the value of underspecifying a logic function so as to allow for selection of the simplest possible implementation. This technique must be used carefully to avoid incorrect behavior, so we illustrate the possibility of misuse with an example, then talk about several ways of solving the example correctly. We conclude by generalizing the ideas in the example to several important application areas and talking about related problems.

Logic Properties

Table 1 (on the next page) lists a number of properties of Boolean logic. Most of these are easy to derive from our earlier definitions, but a few may be surprising to you. In particular, in the algebra of real numbers, multiplication distributes over addition, but addition does not distribute over multiplication. For example, $3 \times (4 + 7) = (3 \times 4) + (3 \times 7)$, but $3 + (4 \times 7) \neq (3 + 4) \times (3 + 7)$. In Boolean algebra, both operators distribute over one another, as indicated in Table 1. The consensus properties may also be nonintuitive. Drawing a K-map may help you understand the consensus property on the right side of the table. For the consensus variant on the left side of the table, consider that since either A or $\overline{A}$ must be 0, either B or C or both must be 1 for the first two factors on the left to be 1 when ANDed together. But in that case, the third factor is also 1, and is thus redundant.

As mentioned previously, Boolean algebra has an elegant symmetry known as a duality, in which any logic statement (an expression or an equation) is related to a second logic statement. To calculate the **dual form** of a Boolean expression or equation, replace 0 with 1, replace 1 with 0, replace AND with OR, and replace OR with AND. *Variables are not changed when finding the dual form.* The dual form of a dual form is the original logic statement. Be careful when calculating a dual form: our convention for ordering arithmetic operations is broken by the exchange, so you may want to add explicit parentheses before calculating the dual. For example, the dual of $AB + C$ is not $A + BC$. Rather, the dual of $AB + C$ is $(A + B)C$. *Add parentheses as necessary when calculating a dual form to ensure that the order of operations does not change.*

Duality has several useful practical applications. First, the **principle of duality** states that any theorem or identity has the same truth value in dual form (we do not prove the principle here). The rows of Table 1 are organized according to this principle: each row contains two equations that are the duals of one another. Second, the dual form is useful when designing certain types of logic, such as the networks of transistors connecting the output of a CMOS gate to high voltage and ground. If you look at the gate designs in the textbook (and particularly those in the exercises), you will notice that these networks are duals. A function/expression is not a theorem nor an identity, thus the principle of duality does not apply to the dual of an expression. However, if you treat the value 0 as "true," the dual form of an expression has the same truth values as the original (operating with value 1 as "true"). Finally, you can calculate the complement of a Boolean function (any expression) by calculating the dual form and then complementing each variable.

Choosing the Best Function

When we specify how something works using a human language, we leave out details. Sometimes we do so deliberately, assuming that a reader or listener can provide the details themselves: "Take me to the airport!" rather than "Please bend your right arm at the elbow and shift your right upper arm forward so as to place your hand near the ignition key. Next, ..."

You know the basic technique for implementing a Boolean function using **combinational logic**: use a K-map to identify a reasonable SOP or POS form, draw the resulting design, and perhaps convert to NAND/NOR gates.

$1 + A = 1$	$0 \cdot A = 0$	
$1 \cdot A = A$	$0 + A = A$	
$A + A = A$	$A \cdot A = A$	
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	
$\overline{A + B} = \overline{A}\ \overline{B}$	$\overline{AB} = \overline{A} + \overline{B}$	DeMorgan's laws
$(A + B)C = AC + BC$	$A\ B + C = (A + C)(B + C)$	distribution
$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$	$A\ B + \overline{A}\ C + B\ C = A\ B + \overline{A}\ C$	consensus

Table 1: Boolean logic properties. The two columns are dual forms of one another.

When we develop combinational logic designs, we may also choose to leave some aspects unspecified. In particular, the value of a Boolean logic function to be implemented may not matter for some input combinations. If we express the function as a truth table, we may choose to mark the function's value for some input combinations as "**don't care**," which is written as "x" (no quotes).

What is the benefit of using "don't care" values? Using "don't care" values allows you to choose from among several possible logic functions, all of which produce the desired results (as well as some combination of 0s and 1s in place of the "don't care" values). Each input combination marked as "don't care" doubles the number of functions that can be chosen to implement the design, often enabling the logic needed for implementation to be simpler.

For example, the K-map to the right specifies a function $F(A, B, C)$ with two "don't care" entries. If you are asked to design combinational logic for this function, you can choose any values for the two "don't care" entries. When identifying prime implicants, each "x" can either be a 0 or a 1.

F	AB 00	01	11	10
C 0	0	1	x	x
C 1	0	1	1	0

Depending on the choices made for the x's, we obtain one of the following four functions:

$$
\begin{aligned}
F &= \overline{A}\ B + B\ C \\
F &= \overline{A}\ B + B\ C + A\ \overline{B}\ \overline{C} \\
F &= B \\
F &= B + A\ \overline{C}
\end{aligned}
$$

F	AB 00	01	11	10
C 0	0	1	*1*	*0*
C 1	0	1	1	0

Given this set of choices, a designer typically chooses the third: $F = B$, which corresponds to the K-map shown to the right of the equations. The design then produces $F = 1$ when $A = 1$, $B = 1$, and $C = 0$ ($ABC = 110$), and produces $F = 0$ when $A = 1$, $B = 0$, and $C = 0$ ($ABC = 100$). These differences are marked with shading and green italics in the new K-map. No implementation ever produces an "x."

Caring about Don't Cares

What can go wrong? In the context of a digital system, unspecified details may or may not be important. However, *any implementation of a specification implies decisions* about these details, so decisions should only be left unspecified if any of the possible answers is indeed acceptable.

As a concrete example, let's design logic to control an ice cream dispenser. The dispenser has two flavors, lychee and mango, but also allows us to create a blend of the two flavors. For each of the two flavors, our logic must output two bits to control the amount of ice cream that comes out of the dispenser. The two-bit $C_L[1:0]$ output of our logic must specify the number of half-servings of lychee ice cream as a binary number, and the two-bit $C_M[1:0]$ output must specify the number of half-servings of mango ice cream. Thus, for either flavor, 00 indicates none of that flavor, 01 indicates one-half of a serving, and 10 indicates a full serving.

Inputs to our logic will consist of three buttons: an L button to request a serving of lychee ice cream, a B button to request a blend—half a serving of each flavor, and an M button to request a serving of mango ice cream. Each button produces a 1 when pressed and a 0 when not pressed.

 ©2012 Steven S. Lumetta. All rights reserved.

Let's start with the assumption that the user only presses one button at a time. In this case, we can treat input combinations in which more than one button is pressed as "don't care" values in the truth tables for the outputs. K-maps for all four output bits appear below. The x's indicate "don't care" values.

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	x	1
1	0	x	x	x

$C_L[0]$

M \ LB	00	01	11	10
0	0	1	x	0
1	0	x	x	x

$C_M[1]$

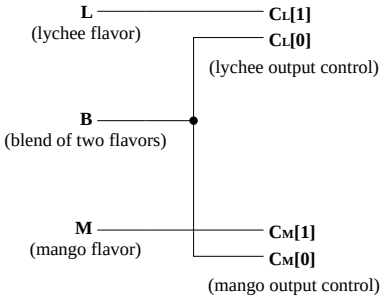
M \ LB	00	01	11	10
0	0	0	x	0
1	1	x	x	x

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	x	0
1	0	x	x	x

When we calculate the logic function for an output, each "don't care" value can be treated as either 0 or 1, whichever is more convenient in terms of creating the logic. In the case of $C_M[1]$, for example, we can treat the three x's in the ellipse as 1s, treat the x outside of the ellipse as a 0, and simply use M (the implicant represented by the ellipse) for $C_M[1]$. The other three output bits are left as an exercise, although the result appears momentarily.

The implementation at right takes full advantage of the "don't care" parts of our specification. In this case, we require no logic at all; we need merely connect the inputs to the correct outputs. Let's verify the operation. We have four cases to consider. First, if none of the buttons are pushed ($LBM = 000$), we get no ice cream, as desired ($C_M = 00$ and $C_L = 00$). Second, if we request lychee ice cream ($LBM = 100$), the outputs are $C_L = 10$ and $C_M = 00$, so we get a full serving of lychee and no mango. Third, if we request a blend ($LBM = 010$), the outputs are $C_L = 01$ and $C_M = 01$, giving us half a serving of each flavor. Finally, if we request mango ice cream ($LBM = 001$), we get no lychee but a full serving of mango.



The K-maps for this implementation appear below. Each of the "don't care" x's from the original design has been replaced with either a 0 or a 1 and highlighted with shading and green italics. Any implementation produces either 0 or 1 for every output bit for every possible input combination.

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	*1*	1
1	0	*0*	*1*	*1*

$C_L[0]$

M \ LB	00	01	11	10
0	0	1	*1*	0
1	0	*1*	*1*	*0*

$C_M[1]$

M \ LB	00	01	11	10
0	0	0	*0*	0
1	1	*1*	*1*	*1*

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	*1*	0
1	0	*1*	*1*	*0*

As you can see, leveraging "don't care" output bits can sometimes significantly simplify our logic. In the case of this example, we were able to completely eliminate any need for gates! Unfortunately, the resulting implementation may sometimes produce unexpected results. Based on the implementation, what happens if a user presses more than one button? The ice cream cup overflows!

Let's see why. Consider the case $LBM = 101$, in which we've pressed both the lychee and mango buttons. Here $C_L = 10$ and $C_M = 10$, so our dispenser releases a full serving of each flavor, or two servings total. Pressing other combinations may have other repercussions as well. Consider pressing lychee and blend ($LBM = 110$). The outputs are then $C_L = 11$ and $C_M = 01$. Hopefully the dispenser simply gives us one and a half servings of lychee and a half serving of mango. However, if the person who designed the dispenser assumed that no one would ever ask for more than one serving, something worse might happen. In other words, giving an input of $C_L = 11$ to the ice cream dispenser may lead to other unexpected behavior if its designer decided that that input pattern was a "don't care."

The root of the problem is that *while we don't care about the value of any particular output marked "x" for any particular input combination, we do actually care about the relationship between the outputs.*

What can we do? When in doubt, it is safest to make choices and to add the new decisions to the specification rather than leaving output values specified as "don't care." For our ice cream dispenser logic, rather than leaving the outputs unspecified whenever a user presses more than one button, we could choose an acceptable outcome for each input combination and replace the x's with 0s and 1s. We might, for example, decide to produce lychee ice cream whenever the lychee button is pressed, regardless of other buttons ($LBM = 1xx$,

which means that we don't care about the inputs B and M, so $LBM = 100$, $LBM = 101$, $LBM = 110$, or $LBM = 111$). That decision alone covers three of the four unspecified input patterns. We might also decide that when the blend and mango buttons are pushed together (but without the lychee button, LBM=011), our logic produces a blend. The resulting K-maps are shown below, again with shading and green italics identifying the combinations in which our original design specified "don't care."

$C_L[1]$

M \ LB	00	01	11	10
0	0	0	*1*	1
1	0	*0*	*1*	*1*

$C_L[0]$

M \ LB	00	01	11	10
0	0	1	*0*	0
1	0	*1*	*0*	*0*

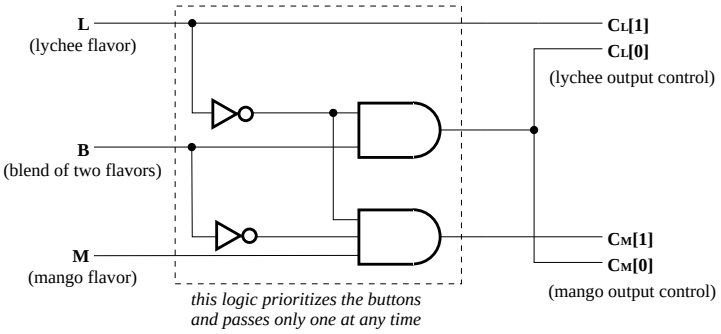
$C_M[1]$

M \ LB	00	01	11	10
0	0	0	*0*	0
1	1	*0*	*0*	*0*

$C_M[0]$

M \ LB	00	01	11	10
0	0	1	*0*	0
1	0	*1*	*0*	*0*

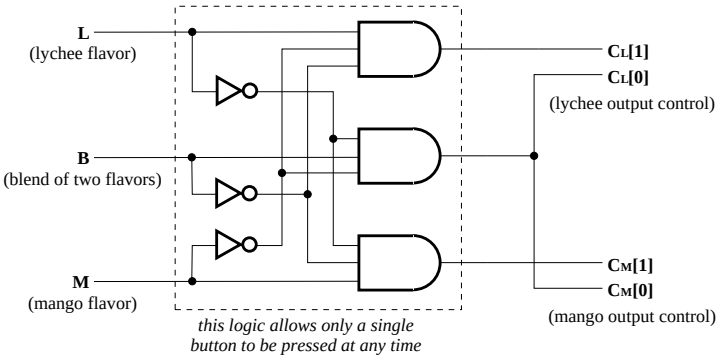
The logic in the dashed box to the right implements the set of choices just discussed, and matches the K-maps above. Based on our additional choices, this implementation enforces a strict priority scheme on the user's button presses. If a user requests lychee, they can also press either or both of the other buttons with no effect. The lychee button has priority. Similarly, if the user does not press lychee, but presses the blend button, pressing the mango button at the same time has no effect. Choosing mango requires that no other buttons be pressed. We have thus chosen a prioritization order for the buttons and imposed this order on the design.



We can view this same implementation in another way. Note the one-to-one correspondence between inputs (on the left) and outputs (on the right) for the dashed box. This logic takes the user's button presses and chooses at most one of the buttons to pass along to our original controller implementation (to the right of the dashed box). In other words, rather than thinking of the logic in the dashed box as implementing a specific set of decisions, we can think of the logic as cleaning up the inputs to ensure that only valid combinations are passed to our original implementation. Once the inputs are cleaned up, the original implementation is acceptable, because input combinations containing more than a single 1 are in fact impossible.

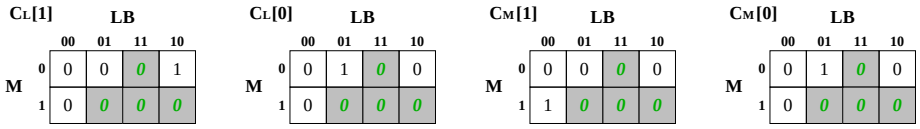
Strict prioritization is one useful way to clean up our inputs. In general, we can design logic to map each of the four undesirable input patterns into one of the permissible combinations (the four that we specified explicitly in our original design, with LBM in the set $\{000, 001, 010, 100\}$). Selecting a prioritization scheme is just one approach for making these choices in a way that is easy for a user to understand and is fairly easy to implement.

A second simple approach is to ignore illegal combinations by mapping them into the "no buttons pressed" input pattern. Such an implementation appears to the right, laid out to show that one can again view the logic in the dashed box either as cleaning up the inputs (by mentally grouping the logic with the inputs) or as a specific set of choices for our "don't care" output values (by grouping the logic with the outputs). In either case, the logic shown enforces our assumptions in a fairly conservative way: if a user presses more than one button, the logic squashes all button presses. Only a single 1 value at a time can pass through to the wires on the right of the figure.



 ©2012 Steven S. Lumetta. All rights reserved.

For completeness, the K-maps corresponding to this implementation are given here.



Generalizations and Applications

The approaches that we illustrated to clean up the input signals to our design have application in many areas. The ideas in this section are drawn from the field and are sometimes the subjects of later classes, but *are not exam material for our class.*

Prioritization of distinct inputs is used to arbitrate between devices attached to a processor. Processors typically execute much more quickly than do devices. When a device needs attention, the device signals the processor by changing the voltage on an interrupt line (the name comes from the idea that the device interrupts the processor's current activity, such as running a user program). However, more than one device may need the attention of the processor simultaneously, so a priority encoder is used to impose a strict order on the devices and to tell the processor about their needs one at a time. If you want to learn more about this application, take ECE391.

When components are designed together, assuming that some input patterns do not occur is common practice, since such assumptions can dramatically reduce the number of gates required, improve performance, reduce power consumption, and so forth. As a side effect, when we want to test a chip to make sure that no defects or other problems prevent the chip from operating correctly, we have to be careful so as not to "test" bit patterns that should never occur in practice. Making up random bit patterns is easy, but can produce bad results or even destroy the chip if some parts of the design have assumed that a combination produced randomly can never occur. To avoid these problems, designers add extra logic that changes the disallowed patterns into allowed patterns, just as we did with our design. The use of random bit patterns is common in Built-In Self Test (BIST), and so the process of inserting extra logic to avoid problems is called BIST hardening. BIST hardening can add 10-20% additional logic to a design. Our graduate class on digital system testing, ECE543, covers this material, but has not been offered recently.