University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Boolean Properties and Optimization

The Dual Form Swaps 0/1 and AND/OR

Boolean algebra has an interesting property called duality.

Let's define the **dual form** of an expression as follows:

- Starting with the expression,
- swap **0** with **1** (just the values, not variables),
- and swap **AND** with **OR**.

Every Boolean Expression Has a Dual Form

For example, what is the dual of

A + (BC) + (0 (D + 1)) ?

First replace the **0** with **1** and the **1** with **0**.

Then replace **+** (**OR**) with **·** (**AND**)
and vice-versa.

We obtain:

A· (B + C)· (1 + (D· 0))

The Dual of the Dual is the Expression

So what is the dual of

$$A \cdot (B + C) \cdot (1 + (D \cdot 0))$$?

Since we're swapping things, swapping them again produces the original expression:

$$A + (BC) + (0\ (D + 1))$$

Thus **any Boolean expression has a unique dual**, and the dual of the dual is the expression (hence the term duality—two aspects of the same thing).

Pitfall: Do Not Change the Order of Operations

Be careful not to change the order of operations when finding a dual form.

For example, the dual form of

$$A + BC$$

is

$$A\,(B + C)$$

The operation on **B** and **C** must happen before the other operation.

Why Do You Care? One Reason: the Principle of Duality

Three reasons:

- CMOS gate structures are dual forms
- Quick way to complement any expression
- the principle of duality

Let's start with the last, which we'll use shortly (when we examine more properties).

Principle of duality: **If a Boolean theorem or identity is true/false, so is the dual of that theorem or identity.**

Generalized DeMorgan is Quick and Easy

Let's say that we have an expression **F**.

To find **F'** … apply DeMorgan's Laws …

Apply repeatedly, as many times as necessary.

Or use the generalized version based on duality:

- Write the dual form of **F**.
- Swap variables and complemented variables.
- (That's all.)

An Example of Finding a Complement with the Dual Form

$$F = AB\,(C + (DL'G(B' + A + E)))\,(H + (J'A'B))$$

What's F'?

The dual is

$$A + B + (C\ (D + L' + G + (B'AE))) + (H\ (J' + A' + B))$$

So

$$F' = A' + B' + (C'\ (D' + L + G' + (BA'E'))) + (H'\ (J + A + B'))$$

You can skip the middle step once you're comfortable with the process.

We Can Derive a Gate's Output from the n-type Network

What about CMOS gate structures?

Think about the network of **n-type** MOSFETS connecting an output **Q** to **0V**.

For example, consider a set of **four n-type arranged in parallel with inputs A, B, C, and D**.

So **Q = 0** if ANY of the transistors is on. In other words, **Q** is **0** when **A + B + C + D**.

Thus **Q = (A + B + C + D)'**. A NOR gate.

We Can Also Derive Function from the p-type Network

What about the **p-type** transistors on the same gate?

- They are arranged in series.
- They connect **Q** to $\mathbf{V_{dd}}$.

But **p-type** transistors are on when their gates are set to **0**. So **Q = 1** when ALL of the inputs are **0**.

Thus **Q = A'B'C'D'**.

That's the same expression, of course.

The Expressions are Related via Generalized DeMorgan

But notice that we can also

- get the second form
- by applying generalized DeMorgan to the first form.

Starting with

Q = (A + B + C + D)',

we find the dual of **A+B+C+D** to be **ABCD**, so

Q = A'B'C'D'.

The Networks are Dual Forms of One Another

The complemented variables come from the use of **p-type** transistors.

The **dual form is built into the gate design**.

If we want to design a gate for something OTHER than NAND, NOR, NOT:

- Write the output as **Q = (expression)'**,
- Build that expression from **n-type** MOSFETs.
- Build the dual of the expression from **p-type** MOSFETs.

An Example of an Unusual Gate

Consider the gate here:

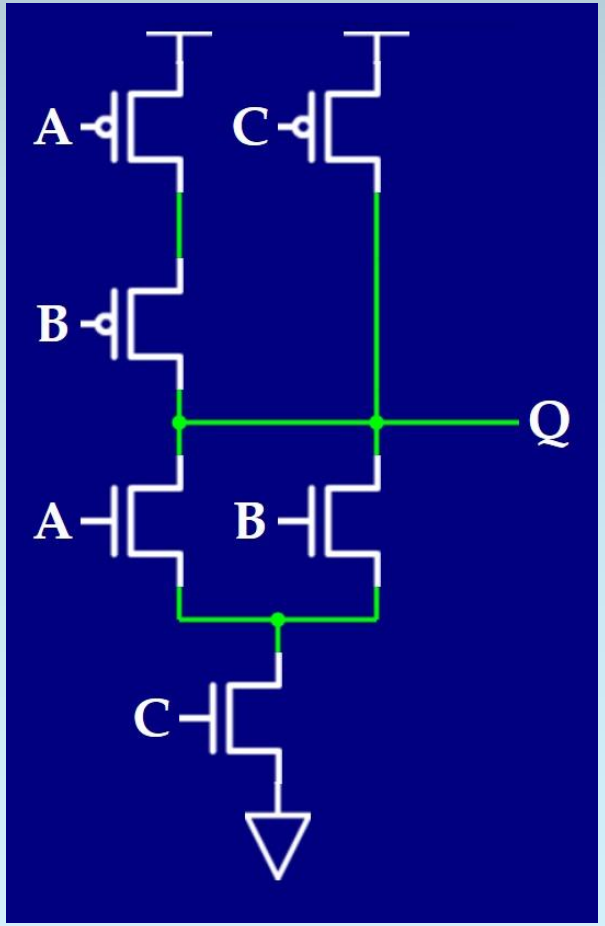
From the **n-type** network,

Q = ((A + B) C)'

The dual of the expression (ignoring the complement) is

AB + C

which is the structure of the **p-type** network.



Area and Speed for the Unusual Gate

So the function **Q = ((A + B) C)'** requires **six transistors and one gate delay**.

We can, of course, limit ourselves to NAND/NOR gates.

In that case, **Q = ((A'B')' C)'**

We use one two-input NAND for **(A'B')'**, and a second two-input NAND for **Q**.

If we assume that **A'** and **B'** are available, **the NAND design requires eight transistors and two gate delays**.

Optimization versus Abstraction

Most designers just use NAND and NOR (or, today, even higher-level abstractions!).

In general:

- breaking abstraction boundaries can give us an advantage,
- but the boundaries make the design task less complex,
- which improves human productivity and reduces the likelihood of mistakes.

That's another tradeoff.

Computer aided design (CAD) tools can perform some of these optimizations for us, too.

Simple Boolean Properties

Easy, but useful to commit to memory for analyzing circuits…

$1 + A = 1$	$0 \cdot A = 0$
$1 \cdot A = A$	$0 + A = A$
$A + A = A$	$A \cdot A = A$
$A \cdot A' = 0$	$A + A' = 1$

(Each row gives two dual forms.)

More Dual Form Boolean Properties

DeMorgan's Laws are also dual forms

(A + B)' = A'B' **(AB)' = A' + B'**

What about distributivity? Here's the rule that you know from our usual algebra

A(B + C) = AB + AC

(multiplication distributes over addition)

It's also true in Boolean algebra:
AND distributes over OR.

OR Also Distributes Over AND in Boolean Algebra

A(B + C) = AB + AC

Now take the dual form…

A + BC = (A + B)(A + C)

OR distributes over AND!

(Note that this property does NOT hold in our usual algebra. $14 + 7 \cdot 4 \neq (14 + 7)(14 + 4)$)

One More Property: Consensus

The last property is non-intuitive.

AB + A'C + BC = AB + A'C

It's called "consensus" because

- the first two terms TOGETHER (when both are true, and thus reach a consensus) imply the third term
- so the third term can be dropped.

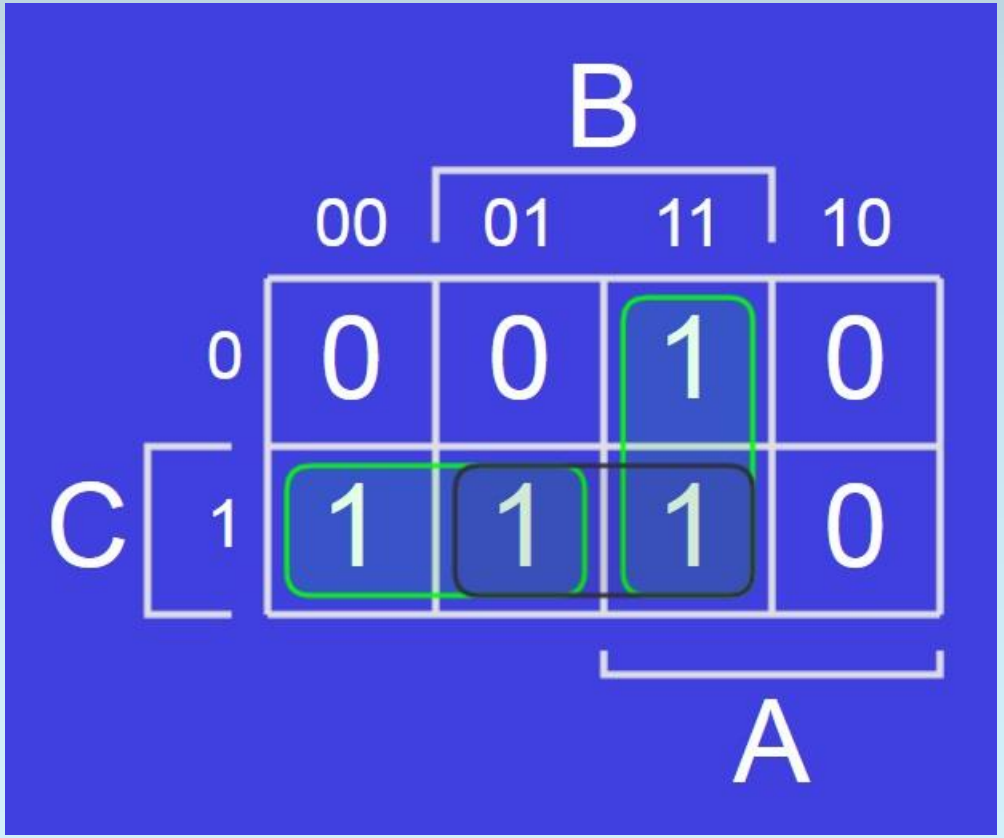
A K-Map Illustrates Consensus Well

Let's look at a K-map.

AB is the vertical green loop.

A'C is the horizontal green loop.

BC is the black loop.



Consensus Has Two Dual Forms (SOP and POS)

And, of course, there is another form of consensus for **POS** form.

Start with our first form:

$$AB + A'C + BC = AB + A'C$$

Then find the dual to obtain:

$$(A + B)(A' + C)(B + C) = (A + B)(A' + C)$$

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Don't Care Outputs

Some Input Combinations May Not Matter

Sometimes, we don't care whether a particular input combination generates a 0 or a 1.

For example,

- when an **input** combination is **impossible to generate**, or
- when **outputs are ignored** in the case of an input combination.

For Such Inputs, Use 'x' to Indicate "Don't Care"

In such cases, we **use 'x'** (called a **"don't care"**) in place of the desired output.

Indicates that either 0 or 1 is acceptable.

However: **whatever we implement will generate a 0 or a 1**, not a "don't care."

So we need to be sure that we really do not care.

Why Are "Don't' Cares" Useful?

More choices often means a "better" answer (for any choice of metric).

Say that you optimize a K-map for a function **F**.

Then you consider several other functions **G**, **H**, and **J**.

If you have to pick one of the four functions (**F**, **G**, **H**, or **J**), the choice can't get worse, since **you can always pick F**, but the best choice may be better than **F**.

N "Don't Cares" Allows 2^N Different Functions

Using x's for outputs means allowing more than one function to be chosen.

Each x can become a 0 or a 1.

So optimizing with **N x's means** choosing from among **2^N possible functions**.

An Example with Two "Don't Cares"

Let's do an example.

The function **F** appears to the right, partially specified.

Let's say that we don't care about the value of **F** when **AB=01**.*

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

*This notation means **A=0** AND **B=1**. You can infer that **AB** in this case does not mean **A** AND **B** because the product **AB** has a single truth value (0 or 1).

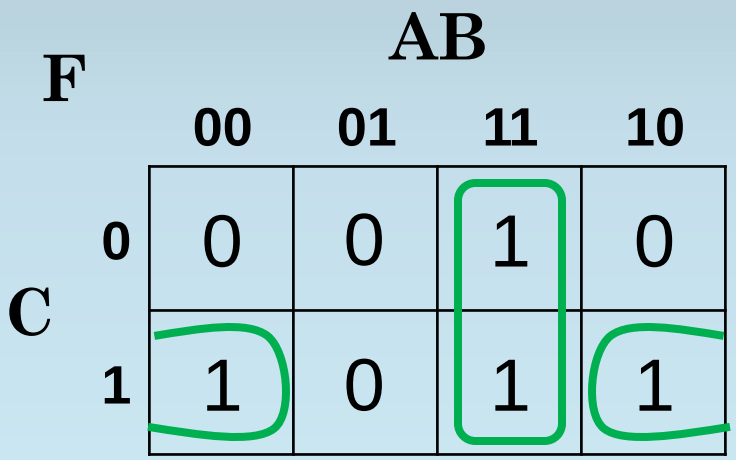
Solution for F with 0s: AB + B'C

One option is to fill the blanks with **0s**.*

Then we can solve.

F = AB + B'C

But we could have chosen values other than **0**, too.



*Without more information about **F**, filling with **0s** is no better nor worse than any other choice.

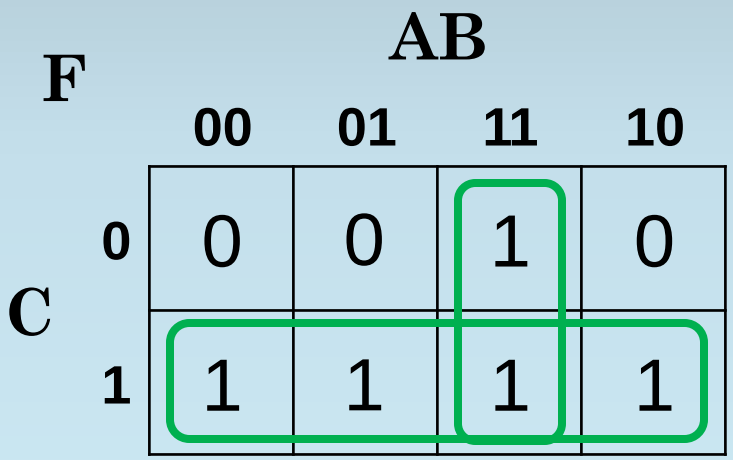
Solution for F with a 0 and a 1: AB + C

For example, we could put a **0** and a **1**…

And then solve.

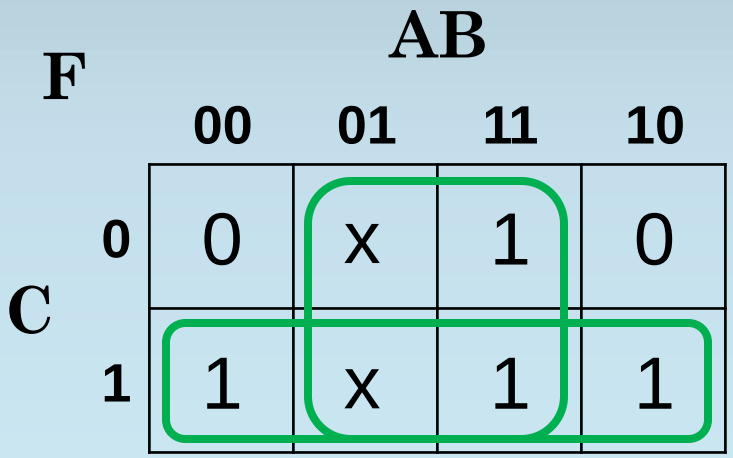
$$F = AB + C$$

This function is better than the first one (it has one fewer literal).



Solution for F with "Don't Cares": B + C

Rather than solving for all four possibilities, let's write **x's** into the K-map.



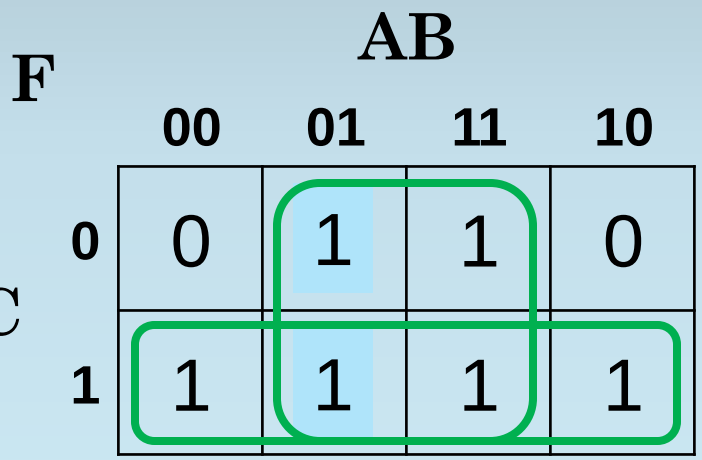
The **x's** can be **0s** or **1s**, so to solve the K-map,

- we **can grow loops to include x's**,
- but we **do not need to cover x's**.

F = B + C (the best possible answer)

Always Check that "Don't Cares" Have No Ill Effects

When designing with **x's**, it's a good habit to verify that the **0s** and **1s** generated in place of **x's** do not cause any adverse effects.



For our function, both **x's** become **1s** because they are inside a loop.

(We don't have any more context for this example, so we are done.)

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Caring About Don't Cares and Glue Logic

Let's Have Some Ice Cream!

Let's build something.

Anyone like ice cream?

Let's build an ice cream dispenser.

Mango and pistachio!

Ice cream for photos courtesy of Annapoorna Stores

Start by Specifying the Inputs and Outputs

inputs: **three buttons**

- **M**(ango): 1 when it's pushed
- **B**(lend): 1 when it's pushed
- **P**(istachio): 1 when it's pushed

outputs: **two 2-bit unsigned numbers**

- **C_M[1:0]**: number of ½ cups of mango
- **C_P[1:0]**: number of ½ cups of pistachio

The User Has Three Choices (and One Non-Choice)

Help fill in the truth table…

Push M, get one cup of mango.

Push B, get ½ cup of each.

Push P, get one cup of pistachio.

Push nothing, get nothing.

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1		
1	0	0	10	00
1	0	1		
1	1	0		
1	1	1		

Fill the Rest with Don't Cares

What about the rest?

Who cares?

Fill with x's.

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1	xx	xx
1	0	0	10	00
1	0	1	xx	xx
1	1	0	xx	xx
1	1	1	xx	xx

We Need to Solve for Each Output Bit

Now we can copy to K-maps. First, $C_M[1]$.

$C_M[1]$

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

$C_M[1] = M$

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1	xx	xx
1	0	0	10	00
1	0	1	xx	xx
1	1	0	xx	xx
1	1	1	xx	xx

Solve the Low Bit of Mango

Next, $C_M[0]$.

$C_M[0]$

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

$C_M[0] = B$

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1	xx	xx
1	0	0	10	00
1	0	1	xx	xx
1	1	0	xx	xx
1	1	1	xx	xx

Solve the High Bit of Pistachio

And $C_P[1]$.

$C_P[1]$

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

$C_P[1] = P$

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1	xx	xx
1	0	0	10	00
1	0	1	xx	xx
1	1	0	xx	xx
1	1	1	xx	xx

Solve the Low Bit of Pistachio

And, finally, $C_P[0]$.

$C_P[0]$

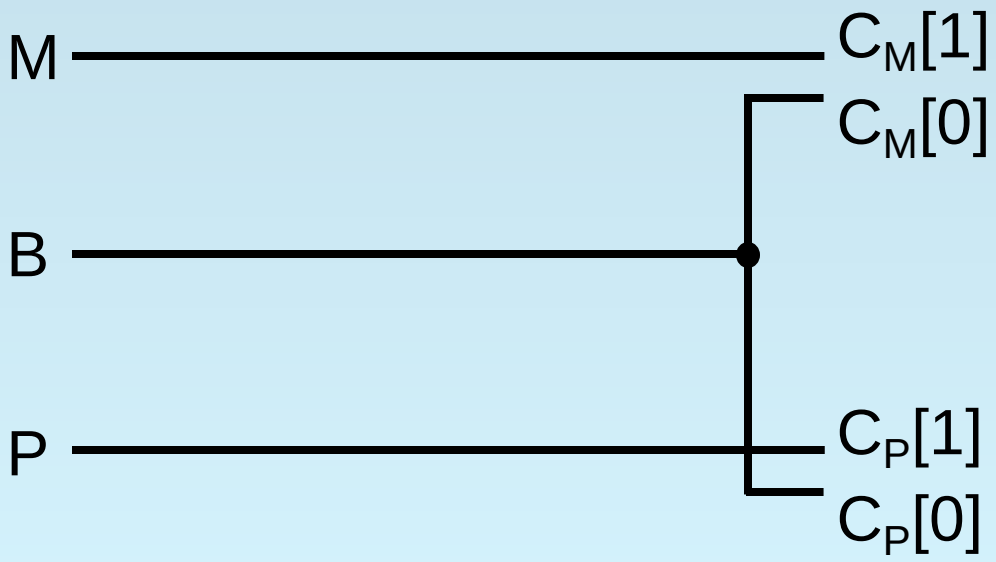
M \ BP	00	01	11	10
0	0	0	x	1
1	0	x	x	x

$C_P[0] = B$

M	B	P	C_M	C_P
0	0	0	00	00
0	0	1	00	10
0	1	0	01	01
0	1	1	xx	xx
1	0	0	10	00
1	0	1	xx	xx
1	1	0	xx	xx
1	1	1	xx	xx

The Solution Requires No Gates!

The don't cares made the functions so simple that **we don't even need any gates**!



What if a User Pushes Two Buttons?

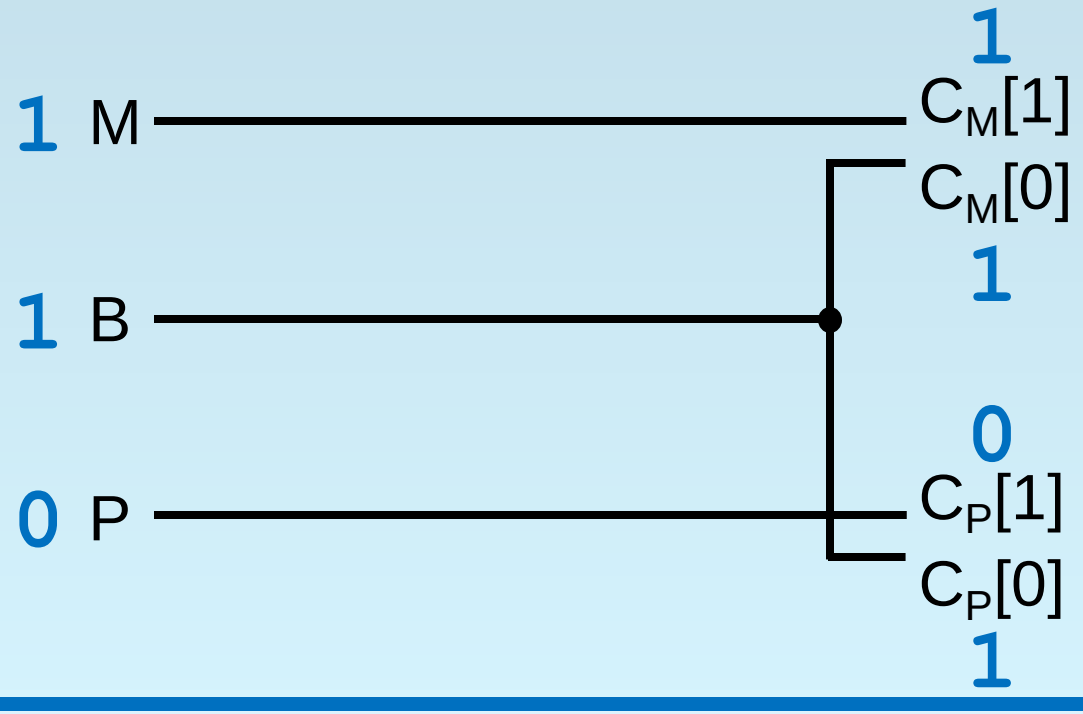
Let's be careful.

What happens if a user **presses M and B** at the same time?

1.5 cups of mango

AND

0.5 cups of pistachio!



Don't Cares: Not for Human Behavior!

In the best case, the cup overflows (2 cups of ice cream instead of 1 cup).

In the worst case,

- the engineer of the mechanical system
- assumed that we would not send 11, and
- something worse happens when we do.

So we DO care.

Generally, **using don't cares when humans are involved is a bad idea**.

Let's Clean Up the Inputs

How can we fix the problem?

One approach:

- **choose specific outputs** for each combination of inputs,
- then solve the K-maps again.

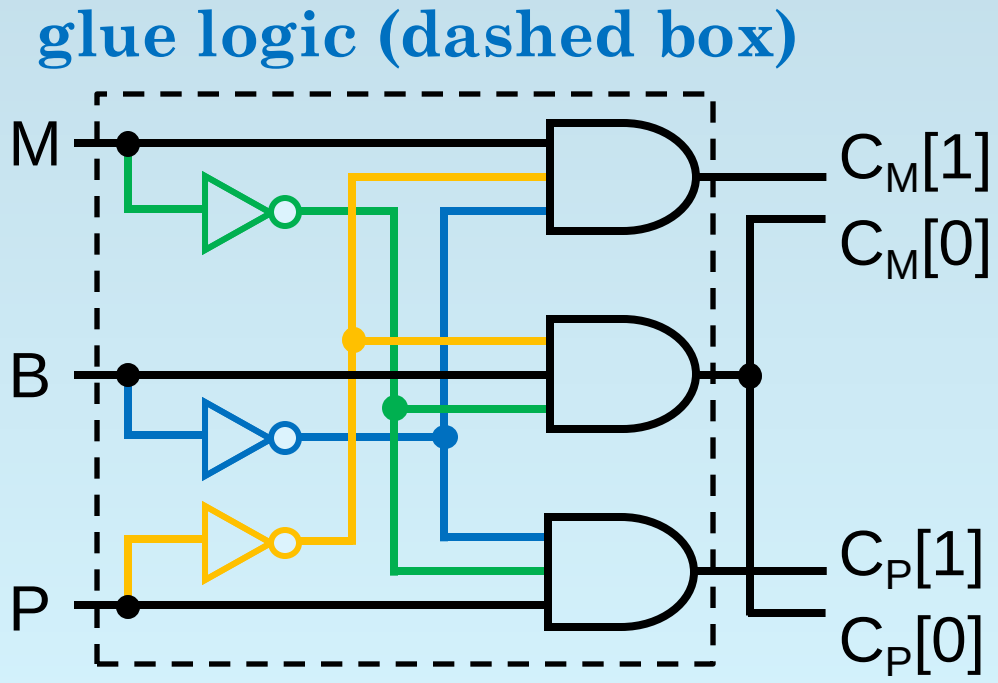
Another approach:

- clean up the inputs with **more logic**
- **prevent** humans from ever producing **bad combinations**.

Use Glue Logic to Ensure that Assumptions Hold

For example, we can force all inputs to zero if the human presses more than one button.

Each of the AND gates produces a 1 iff the corresponding button is the ONLY 1 entered.



The Inputs Can be Cleaned Up in Many Ways

Forcing invalid input combinations to zero is just one strategy.

We could also choose a priority on the buttons (six possible choices).

For example:

- Pistachio overrides other buttons, and
- Mango overrides Blend.

Or use a combination of approaches.

What About Picking Specific K-Maps?

In the case of our ice cream dispenser and the strategy shown, the two approaches are the same (just remove the dashed box!).

In general, however,

- these approaches **vary**
- **in area, speed, and/or power**.

Cleaning up the inputs is perhaps **easier to understand**.