Slide Set 5

for ENEL 353 Fall 2019

Steve Norman, PhD, PEng

Electrical & Computer Engineering
Schulich School of Engineering
University of Calgary

Fall Term, 2019

Contents

Outline of Slide Set 5

Introduction to Karnaugh Maps

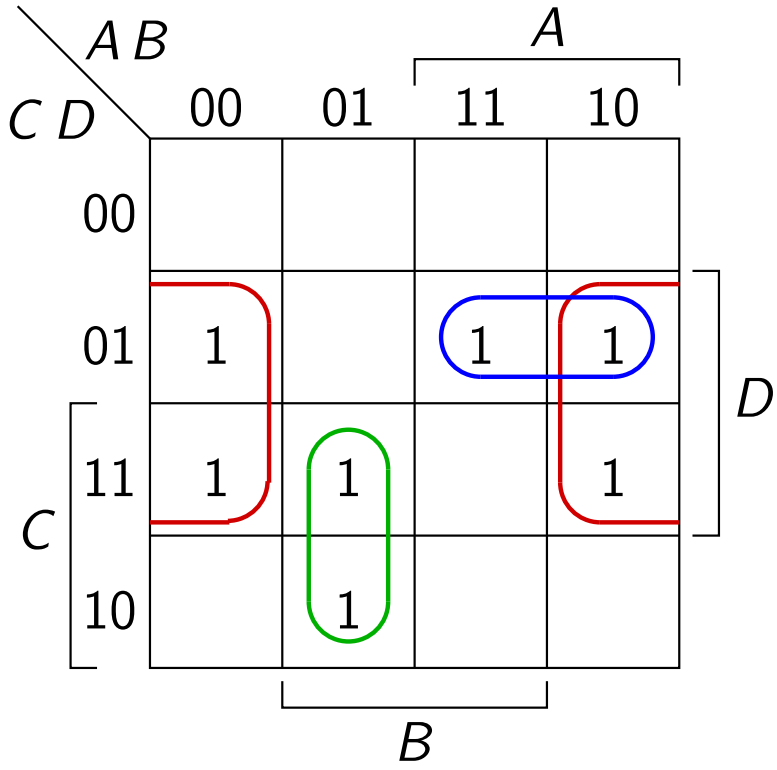
A *Karnaugh map*, often called a "K-map", is a representation of a truth table as a two-dimensional grid of cells—each cell represents a single row of the truth table.

K-maps can be used to quickly find **minimal SOP expressions** for logic functions. A simple trick allows use of K-maps to find **minimal POS expressions** as well.

The next slide is a **preview example** of the use of K-maps. You're not expected to understand all of it at first sight, but you can probably make some good guesses about what is going on ...

A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

K-map for F:



Minimal SOP expression for F:

$$\bar{B}D + \bar{A}BC + A\bar{C}D$$

K-map preliminaries: Gray code ordering

For K-maps to work correctly, rows and columns **must** use Gray code ordering. It's **not an option!**

The one-bit Gray code sequence is trivial: **0, 1.**

The two-bit Gray code sequence is simple, and you should **commit it to memory: 00, 01, 11, 10.** Remember, only one bit can change as you go from one code word to the next.

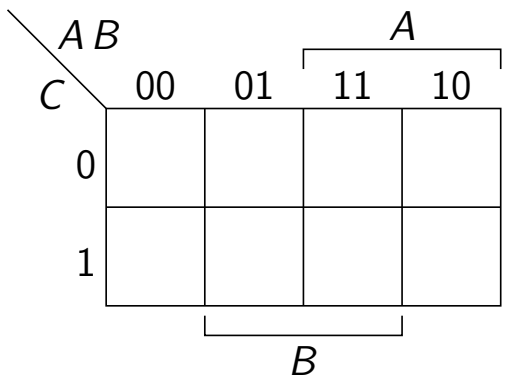
It's easy to slip up, because after you write a lot of 2-input truth tables, the 2-bit unsigned binary code sequence (00, 01, 10, 11) appears in your mind almost automatically.

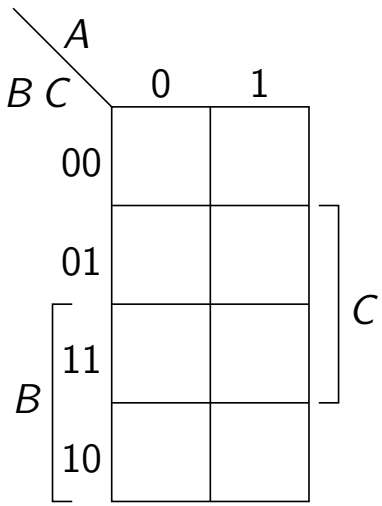
Memorize this:

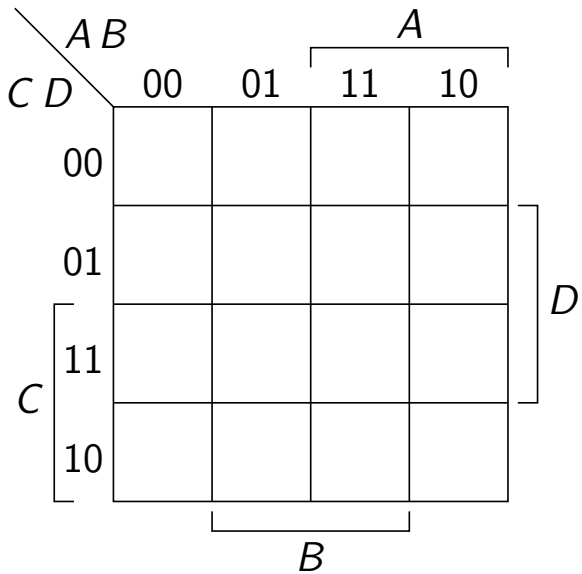
Two-bit Gray code: zero, one, three, two!

Preliminaries: Layout of 3- and 4-variable maps

With 3 variables, you can choose either a "wide" layout or a "tall" layout. With 4 variables, the only reasonable layout is a square of 16 cells.







Let's make some further notes about K-map layouts.

From truth table rows to K-map 1-cells

Each row of the truth table for a function F in which $F = 1$ generates a *1-cell* in the K-map of F. The values of the inputs in a truth table row give you K-map row and column numbers for the 1-cell.

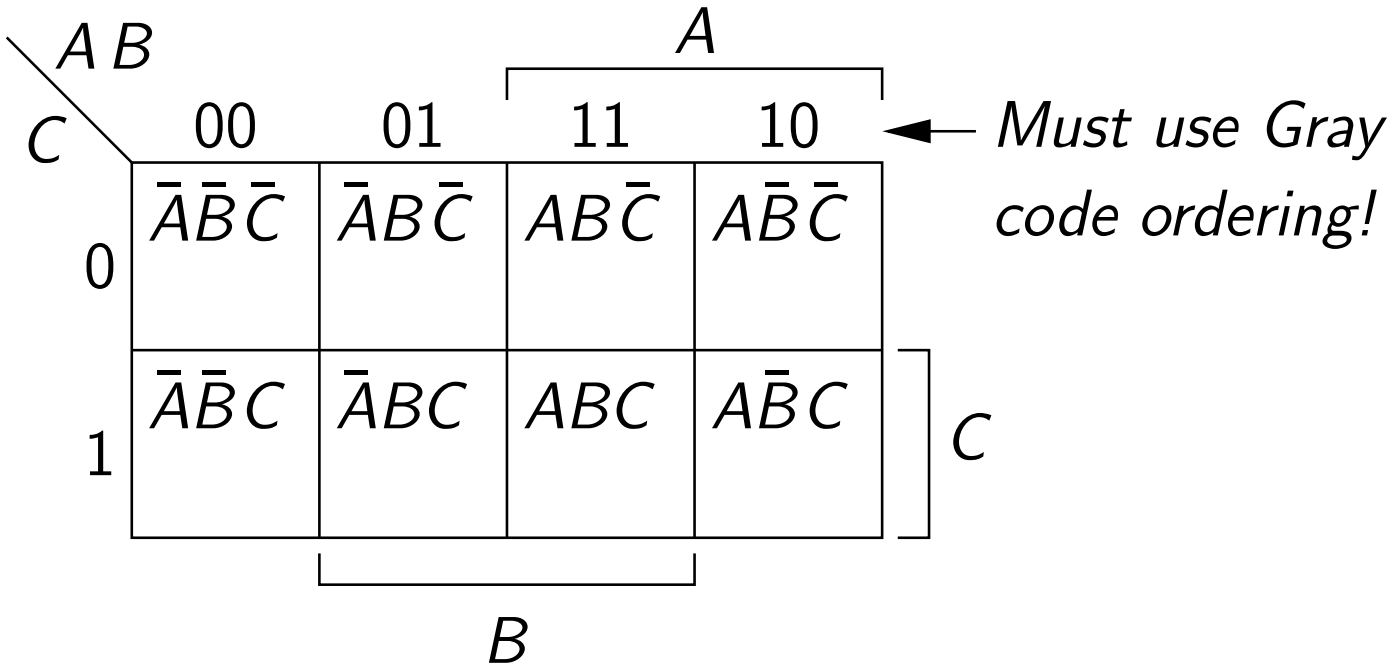
For truth table rows in which $F = 0$, K-map cells are often left **blank**. (Harris & Harris put 0's in those cells.)

Foo	Bar	Quux	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

For the example F on this slide, let's lay out a K-map and mark the 1-cells.

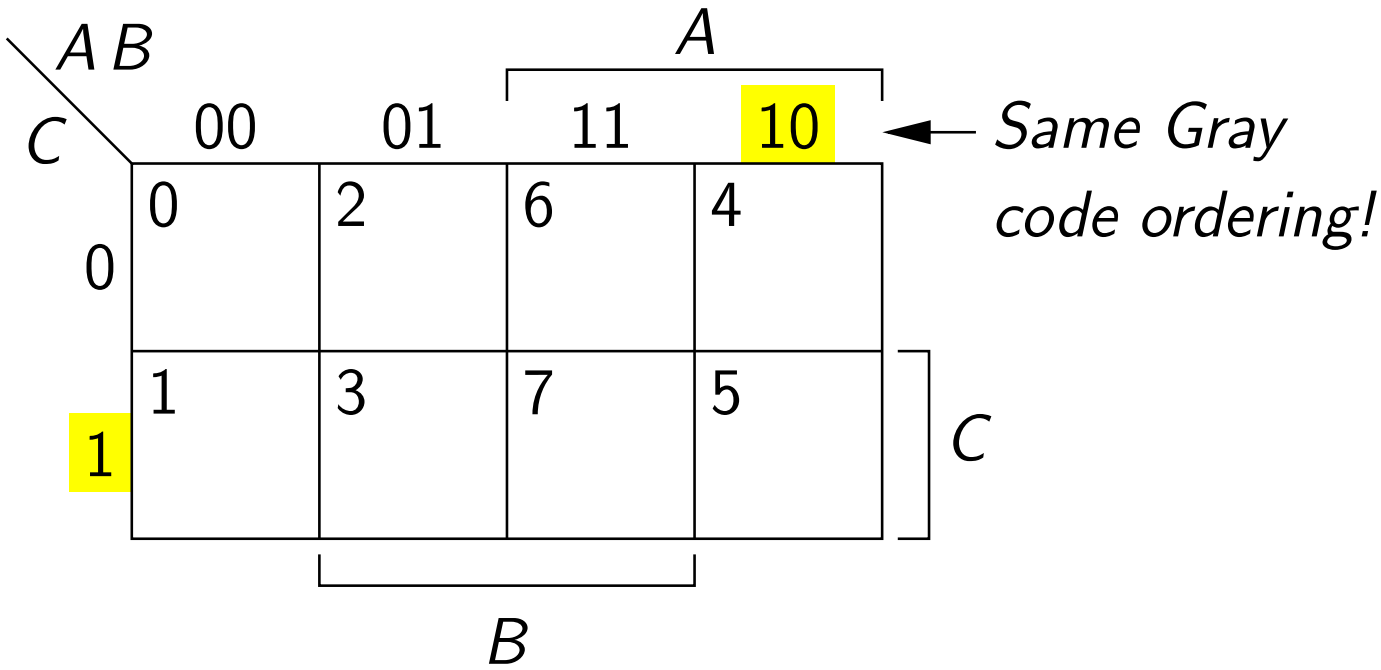
Template for 3-variable K-maps

For $F(A,B,C)$ there are $2^3 = 8$ possible minterms. Each minterm is true for a single truth table row, so we can associate minterms with cells in the K-map ...



Template for 3-variable K-maps, continued

It's often convenient to use minterm numbers instead of writing minterms as products of literals . . .



A quick way to determine the minterm number for a cell: Append the cell's C bit to its AB bits. For example, the lower-right corner cell has minterm number $10\ 1_2 = 5$.

Example 3-variable K-map: carry-out function of a 1-bit full adder

Let's draw a K-map, using the truth table for this function.

A	B	C_{IN}	minterm #	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	2	0
0	1	1	3	1
1	0	0	4	0
1	0	1	5	1
1	1	0	6	1
1	1	1	7	1

Outline of Slide Set 5

Some K-map terminology: *1-cells* and *adjacency*

A *1-cell* is simply a cell with a 1 written into it.

Two cells are *adjacent* if the minterms for the cells differ by only one literal. Examples, for a 3-input function:

- Are the cells for $\bar{A}B\bar{C}$ and $\bar{A}\bar{B}\bar{C}$ adjacent?
- Are the cells for ABC and $A\bar{B}\bar{C}$ adjacent?

A crucial fact about K-maps: **Adjacent cells share either a horizontal or vertical border.**

Important: For "wide" 3-variable maps as shown on the next slide, a shared border includes "wrapping around" from right edge to left edge.

In which K-maps are the 1-cells adjacent?

(A)

C \ AB	00	01	11	10
0	1		1	
1				

(B)

C \ AB	00	01	11	10
0			1	
1		1		

(C)

C \ AB	00	01	11	10
0	1			
1	1			

(D)

C \ AB	00	01	11	10
0				
1		1	1	

(E)

C \ AB	00	01	11	10
0	1			1
1				

(F)

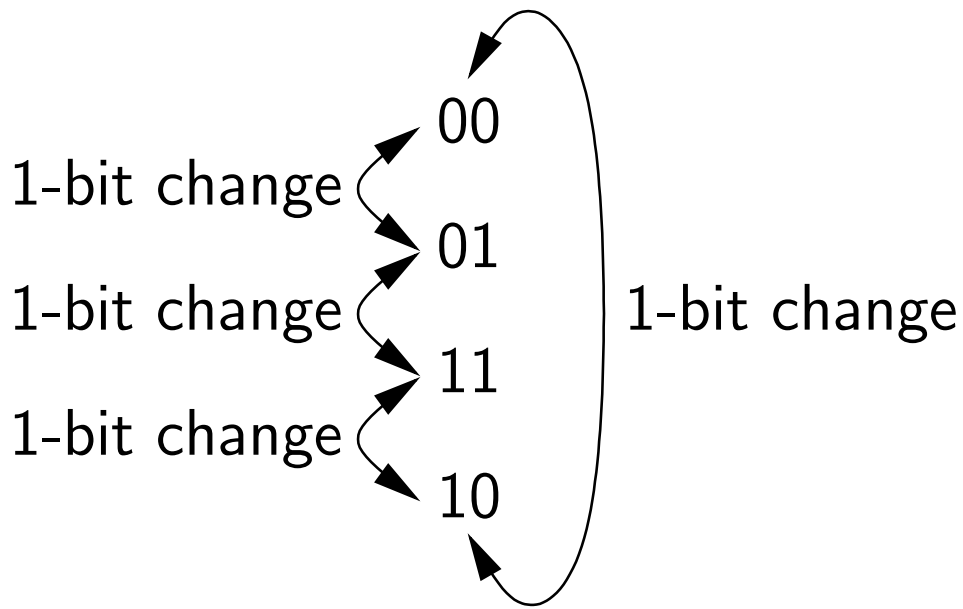
C \ AB	00	01	11	10
0	1			
1				1

Review of Gray code properties

Key property:
Adjacent codes differ in only one bit.

Just as important:
The **first** and **last** codes differ in only one bit.

2-bit Gray code:



Adjacency in algebra and in K-maps

Adjacent minterms differ in exactly one literal. (So, for example, $A\bar{B}C$ and ABC are adjacent, but $A\bar{B}C$ and $\bar{A}BC$ are not.)

Because of the Gray code ordering of cells in a K-map, when crossing from a cell to a neighbouring cell, exactly one bit changes in the cell minterm mumber. In other words, adjacent minterms must occupy adjacent cells.

Let's illustrate this with two examples in a 3-input system:

- $A\bar{B}C$ (minterm 101_2) and ABC (minterm 111_2);
- $A\bar{B}C$ (minterm 101_2) and $\bar{A}BC$ (minterm 011_2).

Theorem T10 of Boolean Algebra: "Combining"

number	theorem	name
T10	$BC + B\bar{C} = B$	Combining
T10'	$(B + C)(B + \bar{C}) = B$	Combining

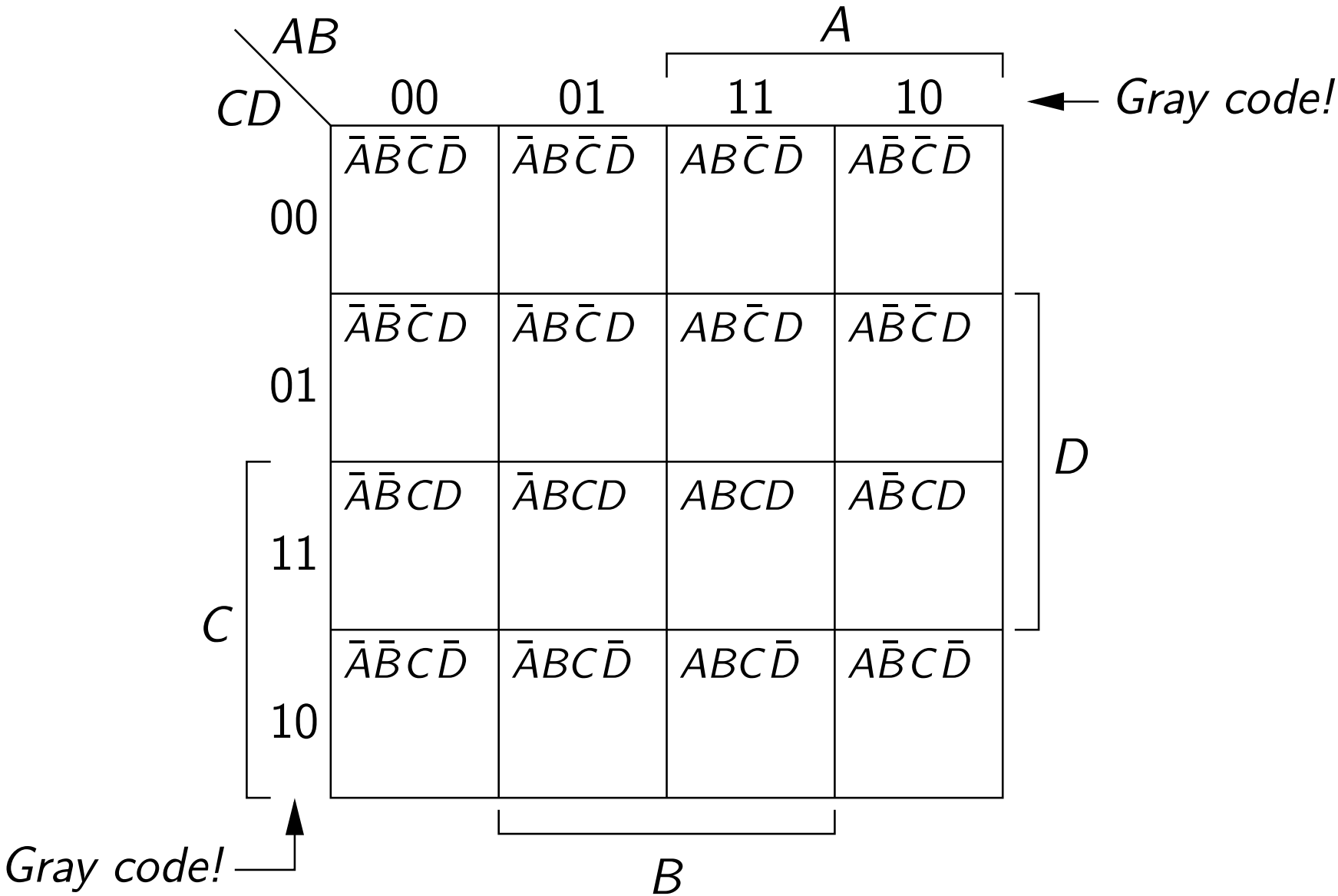
Proof of T10: $B = B \cdot 1 = B(C + \bar{C}) = BC + B\bar{C}$.

This can be used to simplify SOP expressions, for example, $\bar{A}DE + \bar{A}D\bar{E} = \bar{A}D$.

Let's show how this works graphically with a K-map.

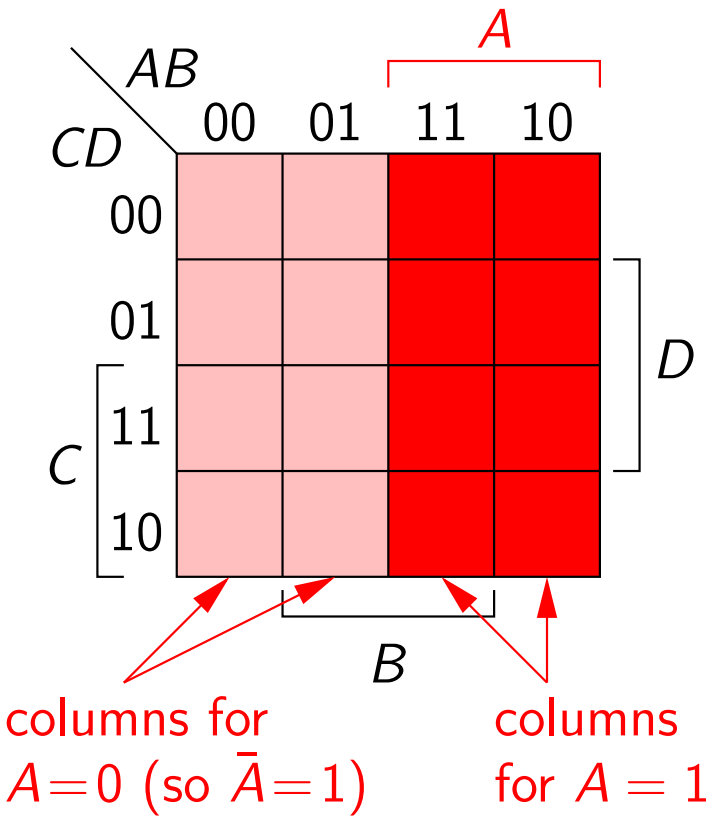
Outline of Slide Set 5

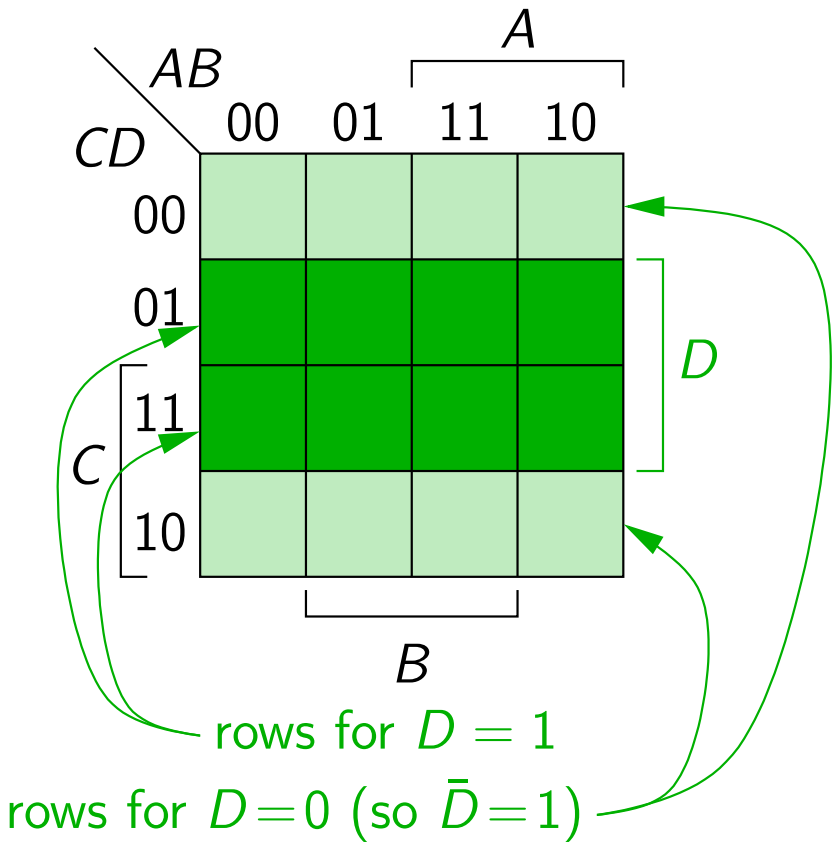
4-variable K-map: template with literals



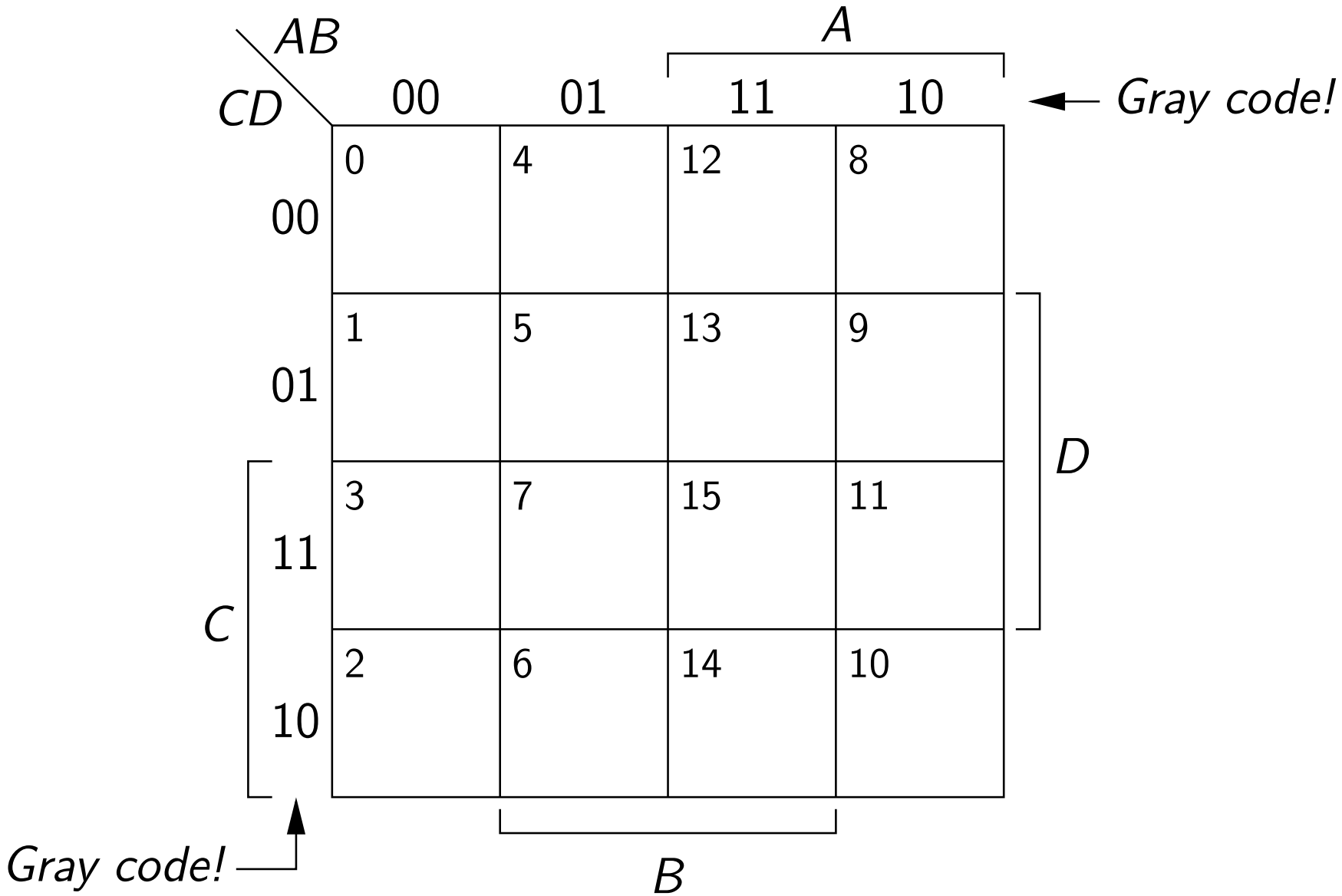
What do the **brackets** outside the K-map mean?

Examples are given for A and D. From those I hope it's also clear what the brackets for B and C indicate.





4-variable K-map: template with minterm numbers



4-variable K-map: example

Let's draw the 4-variable K-map for
$F(A,B,C,D) = \bar{A}B\bar{C}D + A\bar{B}C\bar{D} + ABD.$

Grouping of 1-cells for simplification

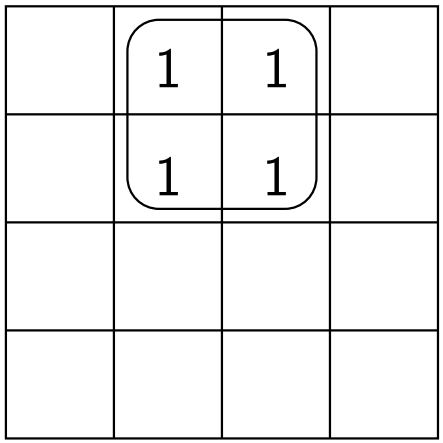
A few slides back we saw:

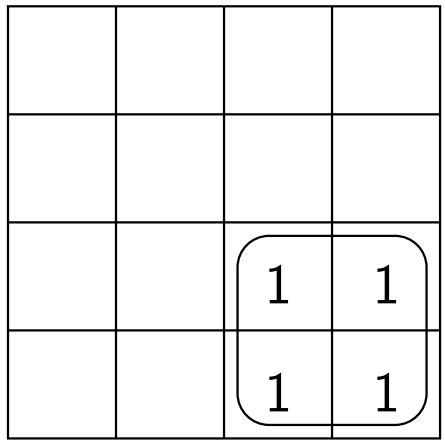
- A **pair** of adjacent 1-cells in a K-map indicates that a pair of minterms can be replaced by a single product with one fewer literal. (Example: $\bar{A}DE + \bar{A}D\bar{E} = \bar{A}D$.)

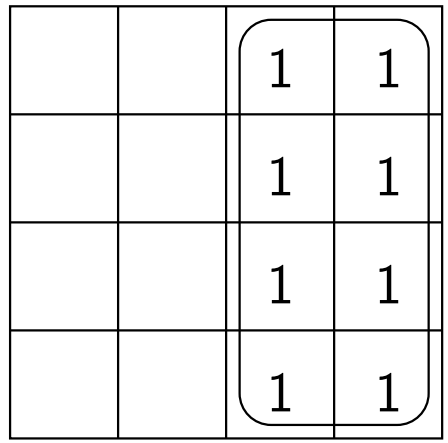
It turns out that this idea **generalizes**:

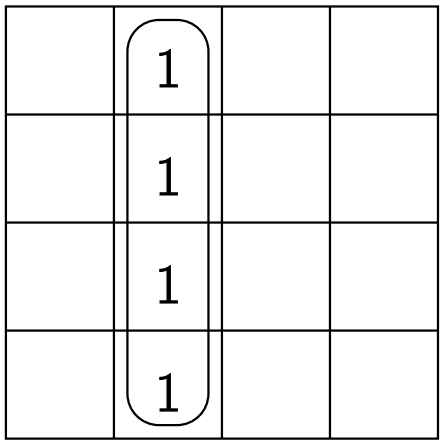
- A **rectangle of four 1-cells** indicates that four minterms can be replaced by a single product with two fewer literals.
- A **rectangle of eight 1-cells** indicates that eight minterms can be replaced by a single product with three fewer literals.

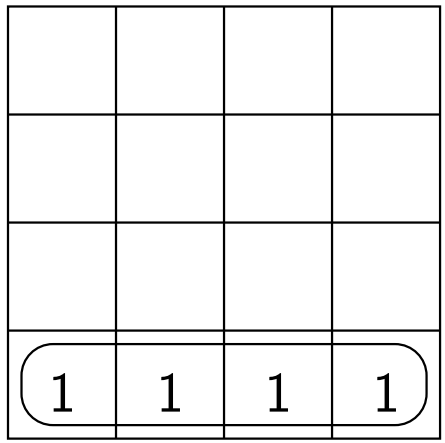
Obvious rectangles of four or eight 1-cells

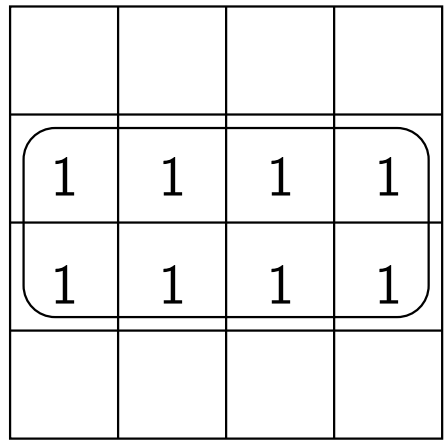




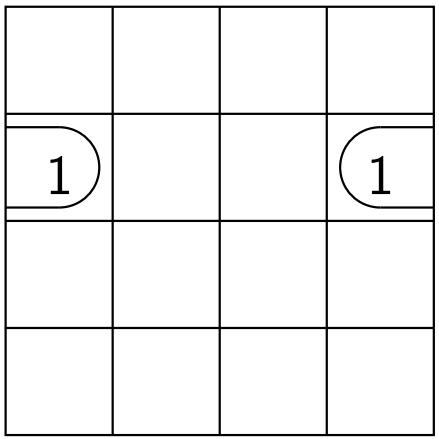




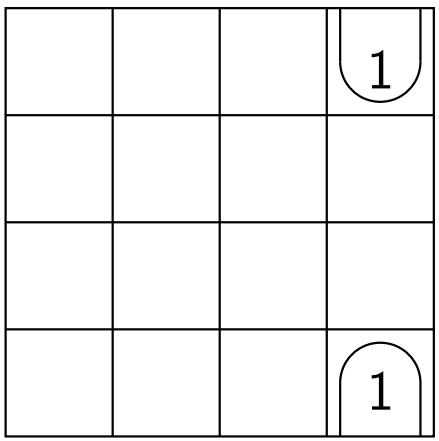




Not-so-obvious rectangles



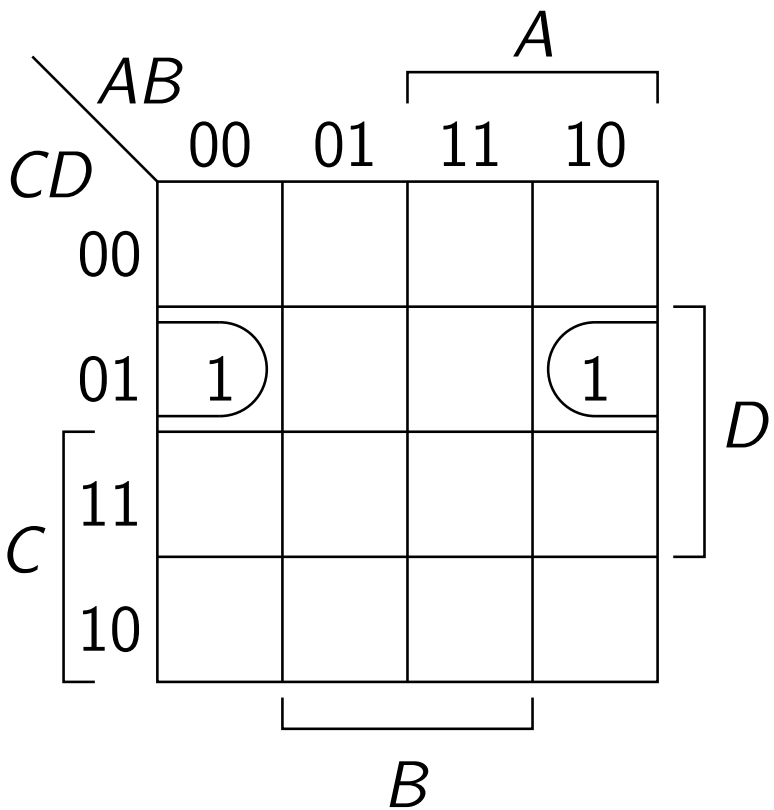
Recall that in a 3-variable K-map with a "wide" layout, you can form a pair by "wrapping" from right to left. In 4-variable maps, you can also wrap from bottom to top.



These kinds of wrapping are also allowed in 4-cell and 8-cell rectangles.

Let's draw some examples of 4-cell and 8-cell rectangles that involve wrapping.

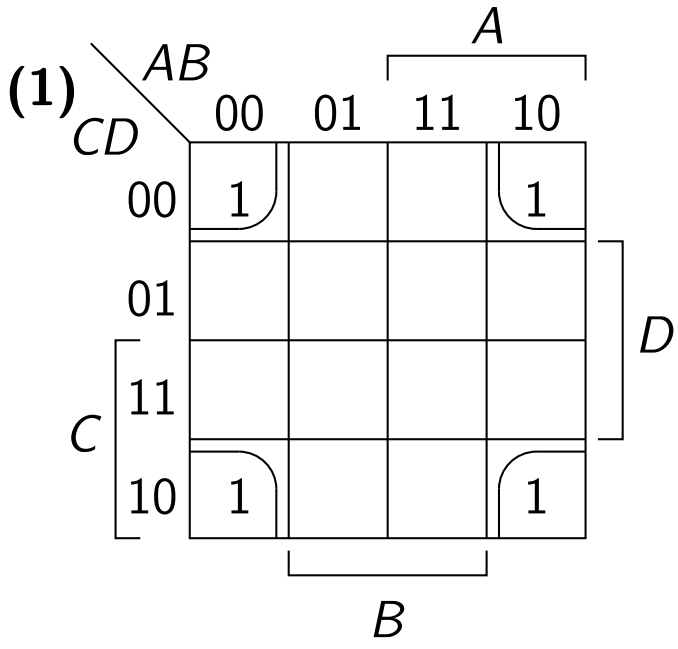
Example: What product does a rectangle represent?

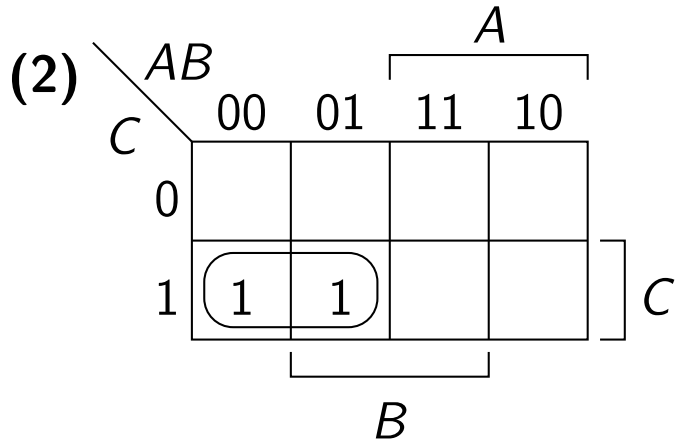


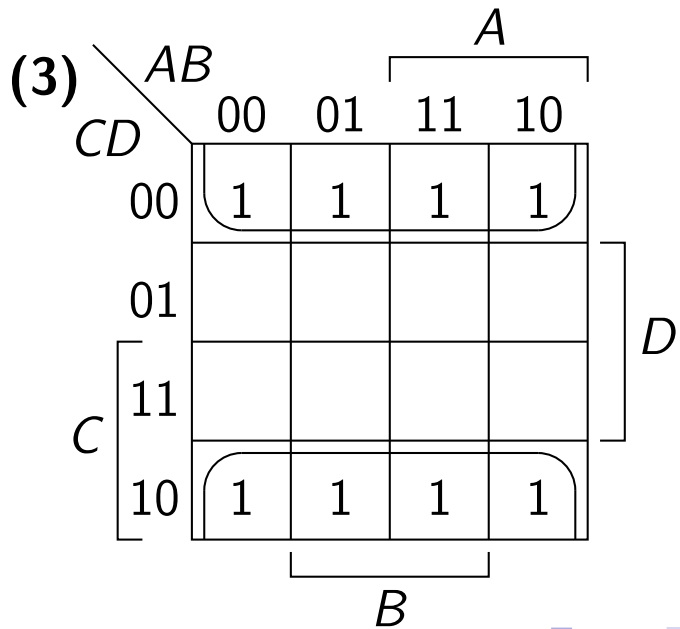
- The rectangle is partly in the "A zone" and partly in the "$\bar{A}$ zone." Include neither A nor $\bar{A}$ in the product.
- It's entirely in the "$\bar{B}$ zone." Include $\bar{B}$.
- It's entirely in the "$\bar{C}$ zone." Include $\bar{C}$.
- It's entirely in the "D zone.". Include D.

Conclusion: The product is $\bar{B}\bar{C}D$.

More examples: What products do these rectangles represent?







Rectangles of three, six, or nine 1-cells are **not helpful**

A rectangle of 2, 4, or 8 1-cells corresponds to a single product in an SOP expression.

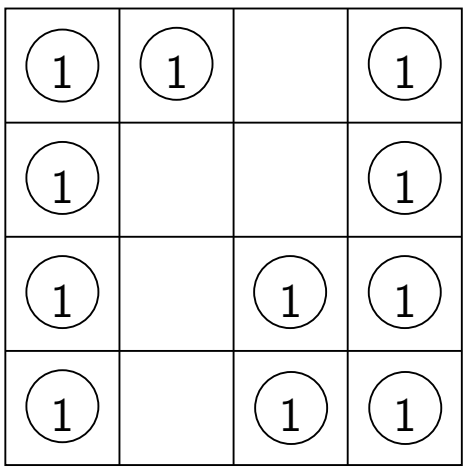
A rectangle of 3, 6, or 9 1-cells **does not** correspond to a single product in an SOP expression, and is therefore **not helpful** in minimization of logic functions.

Outline of Slide Set 5

An **imprecise** description of K-map minimization methods

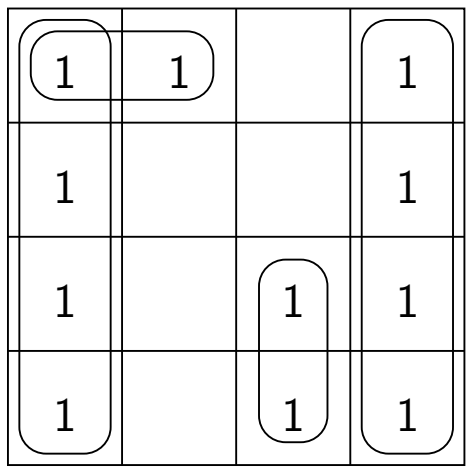
Cover all the 1-cells, and only the 1-cells. Use as few single-cell, 2-cell, 4-cell, and 8-cell rectangles as possible. Make the rectangles as large as possible.

canonical SOP

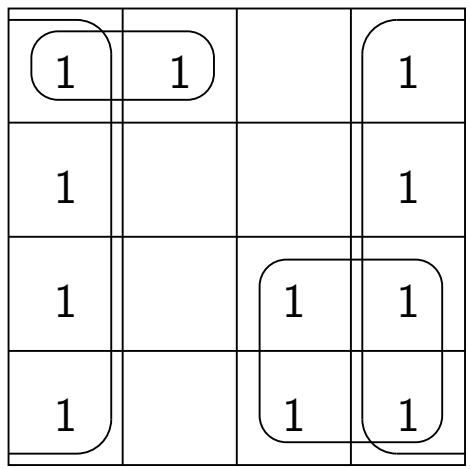


(far from minimal!)

not quite minimal



minimal SOP



Lead-up to an important definition: *Implicant*

"X *implies* Y," means, "Whenever X is true, then Y must also be true." Example: The fact that Joe is in the ICT Building **implies** that Joe is on the U of C campus.

Let's look at a typical SOP expression:
$F = \bar{A}\bar{B}C + \bar{A}B\bar{C} + AC$

Because of the way OR is defined, $\bar{A}\bar{B}C = 1$ implies $F = 1$.

Similarly, $\bar{A}B\bar{C} = 1$ implies $F = 1$, and $AC = 1$ implies $F = 1$.

The products $\bar{A}\bar{B}C$, $\bar{A}B\bar{C}$, and AC are said to be *implicants* of F.

Important definition: *Implicant*

For a given logic function F, the *implicants* of F are all of the products from all of the valid SOP expressions for F.

For example, here is an exhaustive list of valid SOP expressions for the two-input NAND function:

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

So what are all the implicants of this particular F?

Implicants and K-maps

From a K-map for a function F, the implicants of F correspond to

- all of the individual 1-cells;
- all rectangles composed of 2, 4, or 8 1-cells.

(That's for functions of up to four variables—things get more complicated with functions of five or six variables.)

Example 1: *Let's use a K-map to find all the implicants of* $NAND(A,B)$.

Example 2: *Let's work with an example 4-variable K-map, and find a few implicants and SOP expressions.*

Important definition: *Prime implicant*

A *prime implicant* is an implicant that is not fully contained by any other implicant.

Example: If ABC and $AB\bar{C}$ are both implicants of F, they cannot be prime implicants, because they are fully contained by AB.

In a K-map, a prime implicant is a rectangle that cannot be doubled in size without collecting one or more 0-cells.

Illustration of prime and non-prime implicants

Examples, for a 3-variable function and a 4-variable function ...

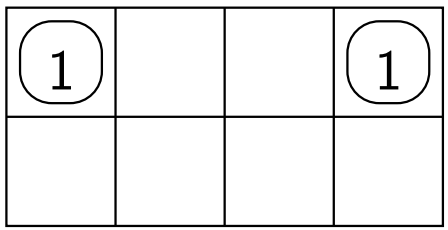
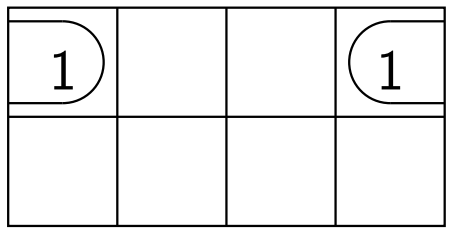
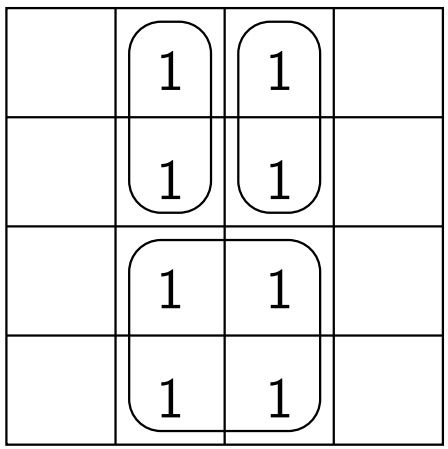
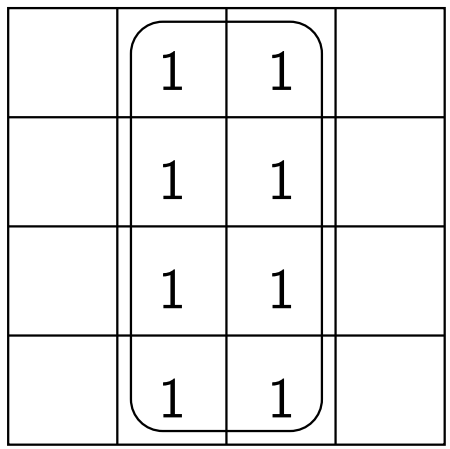
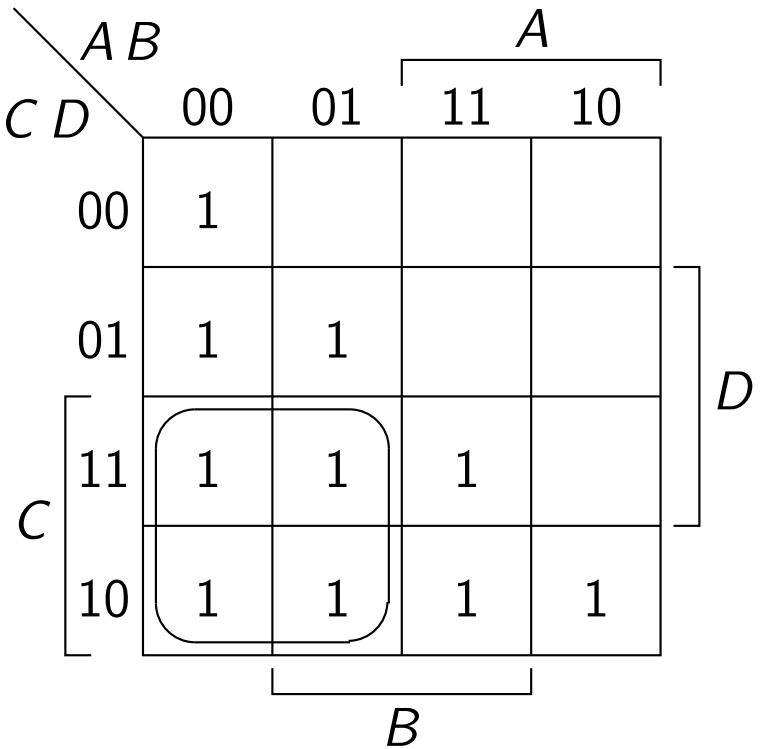
non-prime implicants	**prime implicants**
	
	

Illustration of prime and non-prime implicants, continued



The circled implicant is a prime implicant because it can't be doubled in size without collecting a 0-cell.

What is the product for the circled implicant?

What are all the other prime implicants?

The prime implicant theorem

The theorem says:

> The products in a minimal SOP expression must all be prime implicants.

Sketch of proof:

- Suppose an SOP expression for F includes non-prime implicant X.
- Replace X with a prime implicant that fully contains X.
- The resulting new SOP expression is valid for F and is simpler than the original SOP expression, so the original expression could not have been minimal.

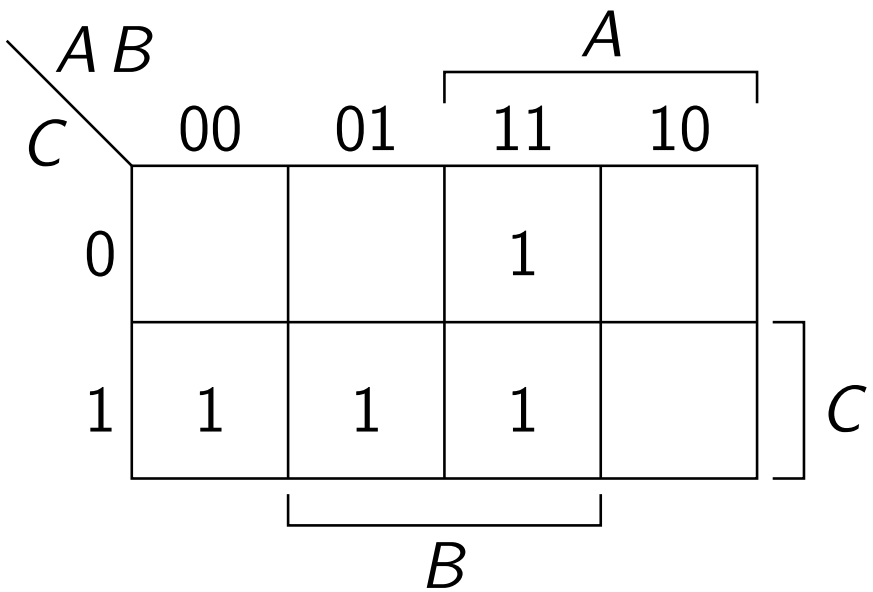
Practical consequence: When you use a K-map to look for a minimal SOP expression, you should **totally ignore** non-prime implicants.

What the prime implicant theorem does **not** say

This is **not generally true**:

> An SOP expression for F in which the products are prime implicants is a minimal SOP expression for F.

Example: *Let's study this K-map . . .*



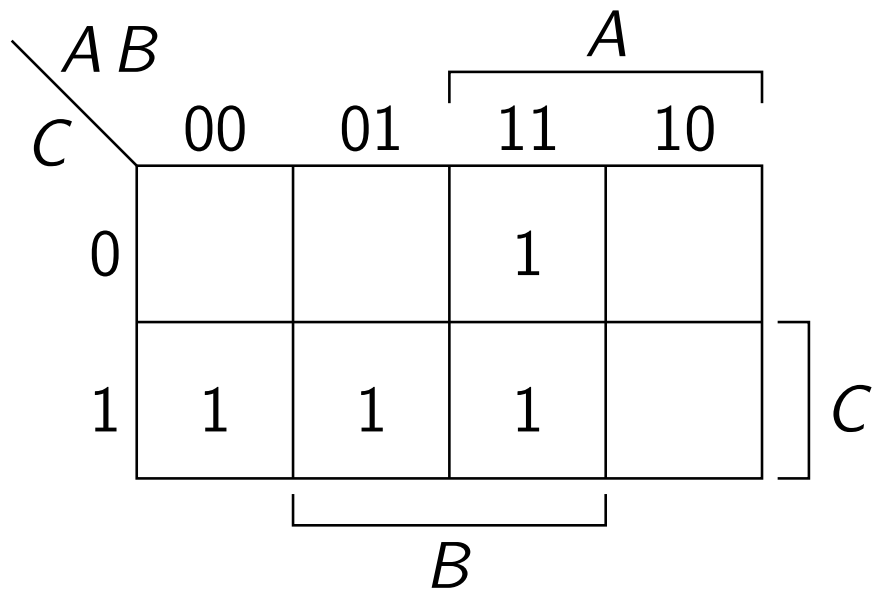
Outline of Slide Set 5

Important definitions: *Distinguished 1-cell* and *essential prime implicant*

A *distinguished 1-cell* of function F is a 1-cell that is covered by exactly one prime implicant of F.

An *essential prime implicant* of F is a prime implicant that covers at least one distinguished 1-cell of F.

Let's study this K-map some more . . .



Distinguished 1-cells, essential prime implicants, and our course textbook

Harris and Harris do not provide definitions of *distinguished 1-cell* or *essential prime implicant*.

However, both concepts are useful for efficient discovery of minimal SOP expressions from K-maps, so in ENEL 353 you **must** know exactly what these terms mean.

(Many other textbooks define *distinguished 1-cell* and *essential prime implicant* exactly as done on the previous slide.)

Essential prime implicants and minimal SOP expressions

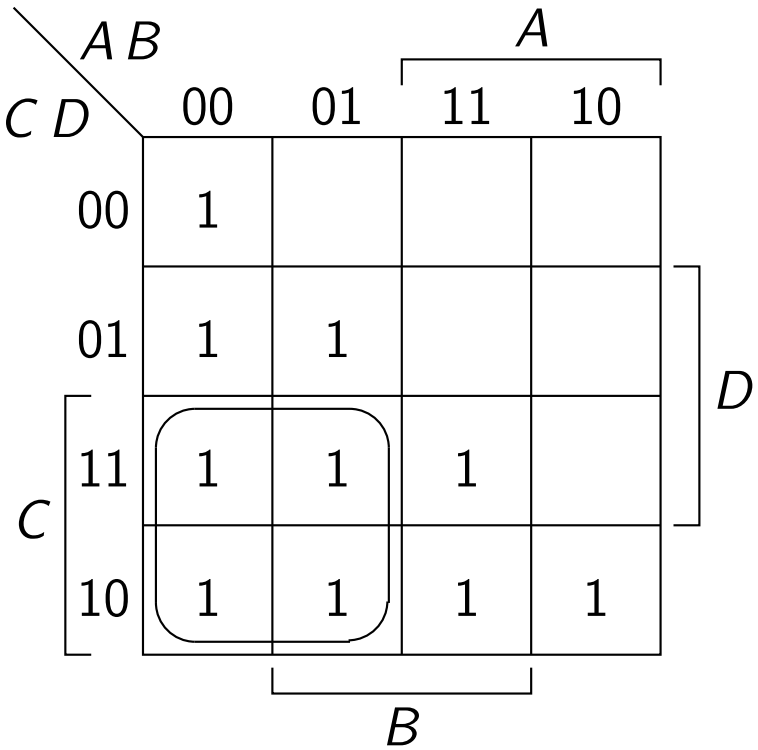
Fact: If the essential prime implicants of F cover all of its 1-cells, the OR of the essential prime implicants is a unique minimal SOP expression for F.

Sketch of proof:

- The minimal SOP expression must be a sum of prime implicants.
- If an essential prime implicant is not used in a sum of prime implicants, at least one 1-cell is not covered, so no essential prime implicant can be left out.

Important consequence: Finally, for at least some functions, we have a way to be **certain** that an SOP expression is a minimal SOP expression!

Review of an earlier example



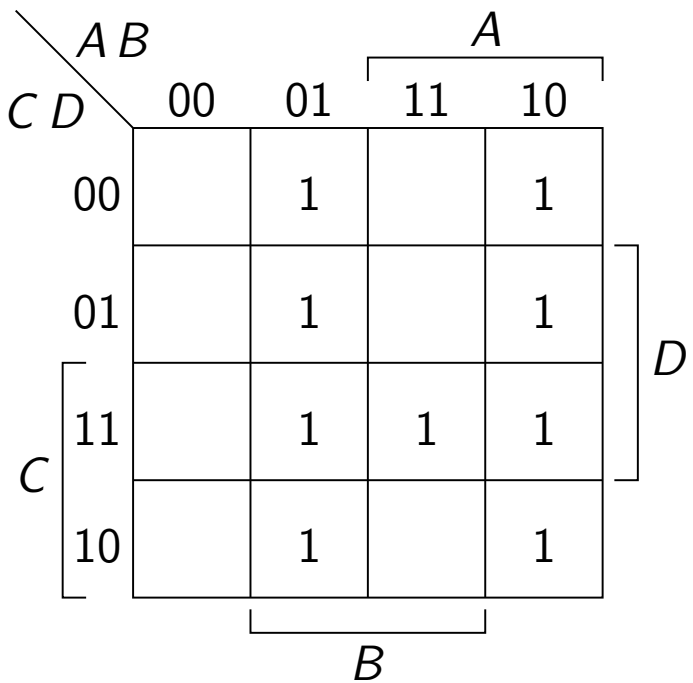
Which prime implicants are essential prime implicants?

Can we use the essential prime implicants to make a minimal SOP expression?

Using only essential prime implicants may fail to cover all the 1-cells

Unfortunately, when looking for minimal SOP expressions, we can't always declare victory after we find all the essential prime implicants.

Let's look at this example . . .



A note about "non-essential" prime implicants

We've just seen that prime implicants in a K-map can be divided into those that are **essential** prime implicants and those that are **not** essential prime implicants.

To call a prime implicant "non-essential" does **not** mean that it is useless or unimportant!

Here is the **correct** distinction ...

- essential PI: contains a distinguished 1-cell, **must** appear in a minimal SOP expression
- non-essential PI: does not contain a distinguished 1-cell, **might or might not** be needed in a minimal SOP expression

Review of important terms

Prime implicant: Group of 1-cells that can't be doubled without collecting a 0-cell.

Distinguished 1-cell: Covered by only one prime implicant.

Essential prime implicant: Covers at least one distinguished 1-cell. **Must** be included in any minimal SOP expression.

Non-essential prime implicant: Does **not** cover any distinguished 1-cells. **Might or might not** need to be included to make a minimal SOP expression.

Prime implicants, essential prime implicants, and minimal SOP expressions

What we know so far ...

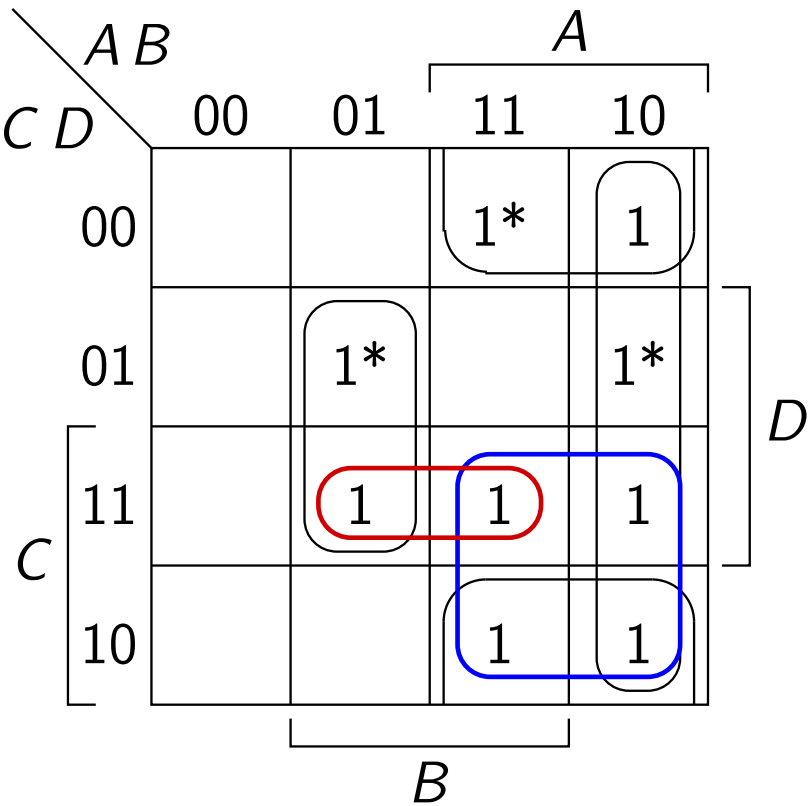
- All implicants in minimal SOP expressions must be prime implicants.
- Essential prime implicants must be included; if not, one or more 1-cells will not be covered.
- For some functions, using only essential prime implicants will fail to cover all the 1-cells.

A method for finding a minimal SOP expression for F from the K-map of F

1. Find all the essential prime implicants.
2. If there are 1-cells not covered by essential prime implicants, then **make the best choice** of non-essential prime implicants to complete the cover.

It's a **method** but not really an **algorithm**, because we don't have a precise definition for "make the best choice". **Reasoning** may be required to justify the choice of non-essential prime implicants.

Example 1

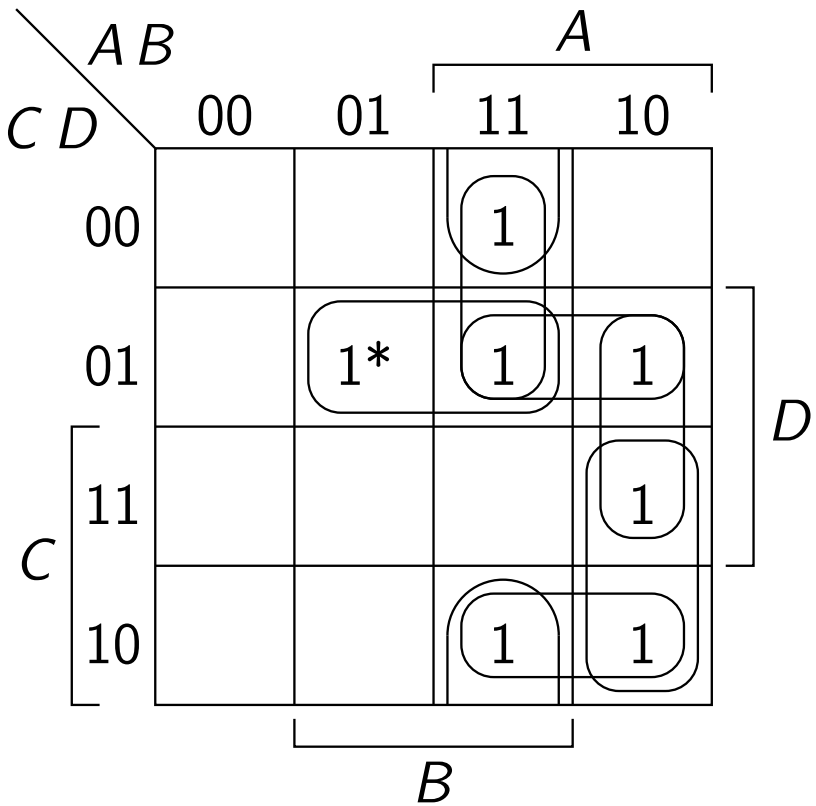


Somebody has found all the PI's and identified the EPI's for us.

$\bar{A}BD + A\bar{B} + A\bar{D}$, the sum of EPI's, does not cover all the 1-cells.

How can we use the K-map to finish the job of finding a minimal SOP expression?

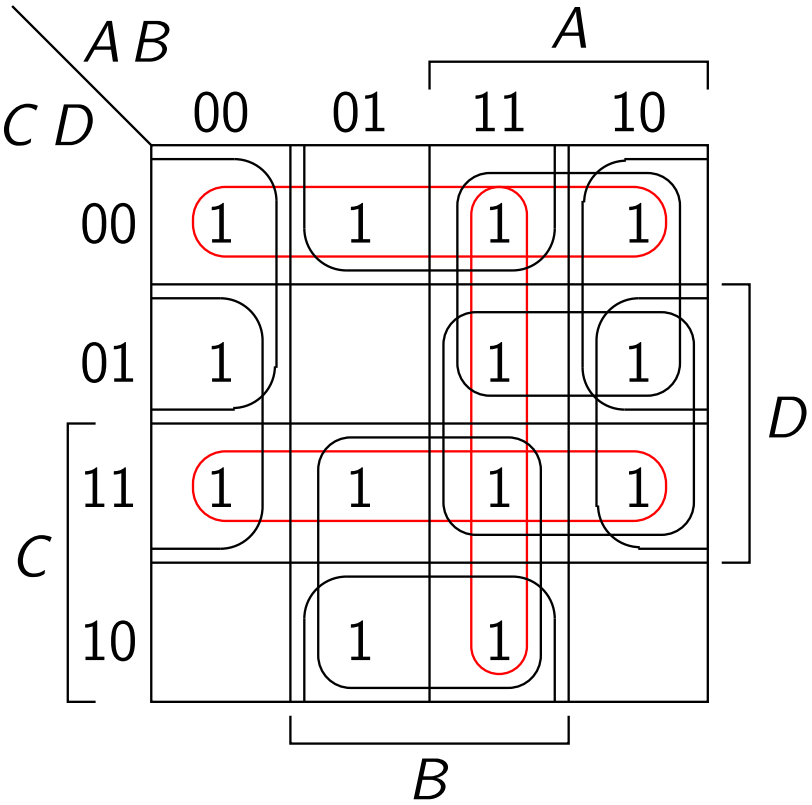
Example 2



Yikes! There is only one essential PI, and there are six non-essential PI's.

How can we use the K-map to find a minimal SOP expression?

Example 3



There are nine prime implicants, all groups of four cells. There are **no** essential prime implicants.

How can we use the K-map to find a minimal SOP expression?

This is Example 3.18 from Marcovitz A. B., *Introduction to Logic Design, 3rd ed.*, 2010, McGraw-Hill. (That's the textbook used for ENEL 353 in Fall 2012 and a few earlier years.)

Outline of Slide Set 5

Don't-care outputs and K-maps

In some truth tables, for some—not all—input combinations, we **don't care** whether a particular output is 0 or 1.

Don't-care output values are marked with X instead of 0 or 1. (Remember that X for don't care in a truth table or K-map does **not** mean the same thing as X for unknown/illegal value at a circuit node.)

Don't-care outputs often help with simplification of SOP expressions.

K-map methods are easy to modify to take don't-cares into account.

A very simple K-map-with-don't-cares example

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	X
1	1	1	X

In normal operation of this particular circuit, we've been told that the input will **never** be $(A,B,C) = (1,1,0)$ or $(A,B,C) = (1,1,1)$, so it **doesn't matter** what F is in the last two rows of the table.

Let's draw K-maps to see how the don't-cares can be used to simplify circuit design.

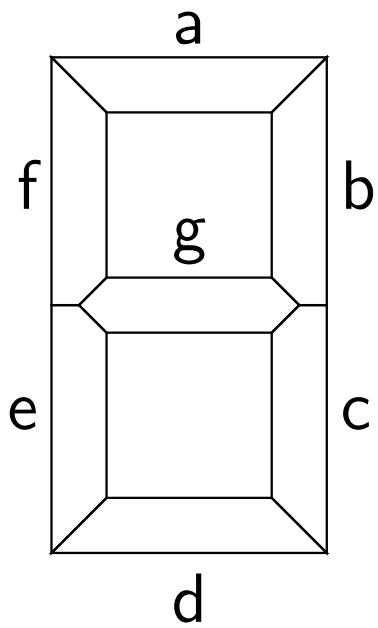
Don't-cares, prime implicants, and essential prime implicants

Our terms need to be adapted *very slightly* to account for don't-cares . . .

- *Prime implicant*: Group of 1, 2, 4, 8, etc., 1-cells **and/or X-cells**; group can't be doubled in size without collecting 0-cells.
- *Distinguished 1-cell:* Same as before. Note that an X-cell cannot be a distinguished 1-cell.
- *Essential prime implicant*: Same as before.

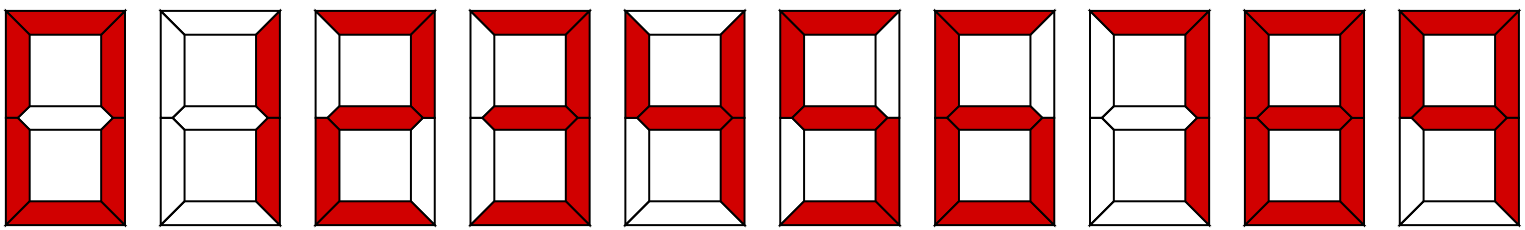
Let's identify PI's, distinguished 1-cells, and EPI's for our earlier don't-care example.

The seven-segment display

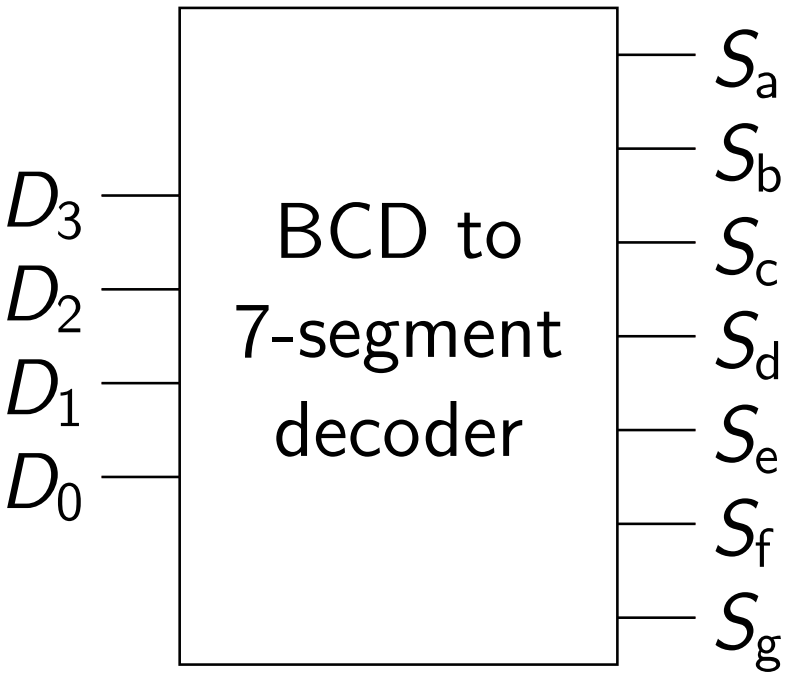


Letters a, b, c, d, e, f, and g identify the seven segments.

With some segments ON and other segments OFF, the array of segments displays one of ten decimal digits . . .



BCD to seven-segment decoder design



For input values 0000, 0001, . . . , 1000, 1001, this combinational circuit must turn segments on or off to display digits 0, 1, . . . , 8, 9.

But what should the circuit do for input bit patterns 1010, 1011, . . . , 1111, which don't correspond to decimal digits? *Let's describe two of several reasonable design decisions that could be made.*

Below is a truth table for two S_b functions, one for each of the design decisions from the previous slide. *Let's find minimal SOP expressions for each* of the S_b *functions.*

					S_b	
D_3	D_2	D_1	D_0	BCD value	design 1	design 2
0	0	0	0	0	1	1
0	0	0	1	1	1	1
0	0	1	0	2	1	1
0	0	1	1	3	1	1
0	1	0	0	4	1	1
0	1	0	1	5	0	0
0	1	1	0	6	0	0
0	1	1	1	7	1	1
1	0	0	0	8	1	1
1	0	0	1	9	1	1
1	0	1	0	n/a	0	X
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	n/a	0	X

Completing the BCD to seven-segment decoder design

Of course, in addition to an SOP expression for S_b, SOP expressions would be needed for S_a, S_c, S_d, S_e, S_f, and S_g.

Harris and Harris give K-maps for S_a using first the "display blank for invalid input" policy, then later the "don't care about invalid input" policy.

Finding SOP expressions for S_c, S_d, S_e, S_f, and S_g is left as an exercise (Exercise 2.34).

Don't-care inputs in truth tables

Don't-care values on the **input** side of a truth table **do not** help with simplification of expressions using K-maps. Instead, they sometimes provide a way to "compress" information in a truth table, as shown in this example . . .

A	B	C	F_1	F_0
0	0	0	0	0
0	0	1	0	1
0	1	X	1	0
1	X	X	1	1

A	B	C	F_1	F_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Let's write out in words what the X's in the example mean.

Outline of Slide Set 5

Minimal POS expressions

First, some **review**. Here is the precise definition for a *minimal sum-of-products expression* for a function F:

- Among all possible SOP expressions for F, none have fewer products than a minimal SOP expression.
- Among all the possible SOP expressions for F that have the same number of products as a minimal SOP expression, none use fewer literals.

Note that a literal counts each time that it is used, so, for example, $A\bar{B} + AC$ has four literals, not three.

Let's use the above material to write a precise definition for the term *minimal product-of-sums expression.*

Minimal SOP, minimal POS, and two-level logic

Minimal SOP is well suited to **two-level AND-OR** circuit implementations of logic functions. Assuming that all input variables for a given problem are available in both true and complementary form, a minimal SOP expression results in an AND-OR circuit with:

- the smallest possible number of AND gates;
- the smallest possible total count of AND gate inputs;
- an OR gate with the smallest possible number of inputs.

What would use of a **minimal POS** *expression give us, in terms of a* **two-level OR-AND** *circuit for a logic function?*

From SOP for $\overline{F}$ to POS for F

Suppose we're looking for a POS expression for a function F.

An easy way to solve the problem is to start with an SOP expression for $\overline{F}$, then apply DeMorgan's Theorem.

Let's do some simple examples:

- $\overline{F(A,B,C,D)} = A\bar{B} + \bar{A}CD$. *Find a POS expression for* F.
- $\overline{F(A,B,C)} = \bar{A}C + B$. *Find a POS expression for* F.

Minimal SOP for $\overline{F}$ leads to minimal POS for F

Suppose we know a minimal SOP expression for $\overline{F}$, where F is some Boolean function.

Then it's a **fact** that we can get a minimal POS expression for F by applying DeMorgan's Theorem to the SOP expression.

Sketch of proof of **fact:**

- DeMorgan's Theorem is a **theorem**. So we know we'll get a **valid** POS expression for F.
- Suppose the POS expression is not minimal—then there must be some better POS expression for F. We could apply DeMorgan to the better POS expression and get a better SOP expression for $\overline{F}$ than the SOP expression we started with.
  That's impossible—we know that the original SOP expression was minimal SOP. So the POS expression we got from applying DeMorgan must be minimal!

We can apply this **fact** to get minimal POS expressions using K-maps . . .

Outline of Slide Set 5

K-maps and minimal POS expressions

To find a minimal POS expression for F using a K-map, there is a simple procedure.

- Make a K-map for $\overline{F}$.
- Use the K-map to find a minimal SOP expression for $\overline{F}$.
- Apply DeMorgan's Theorem to get a minimal POS expression for F.

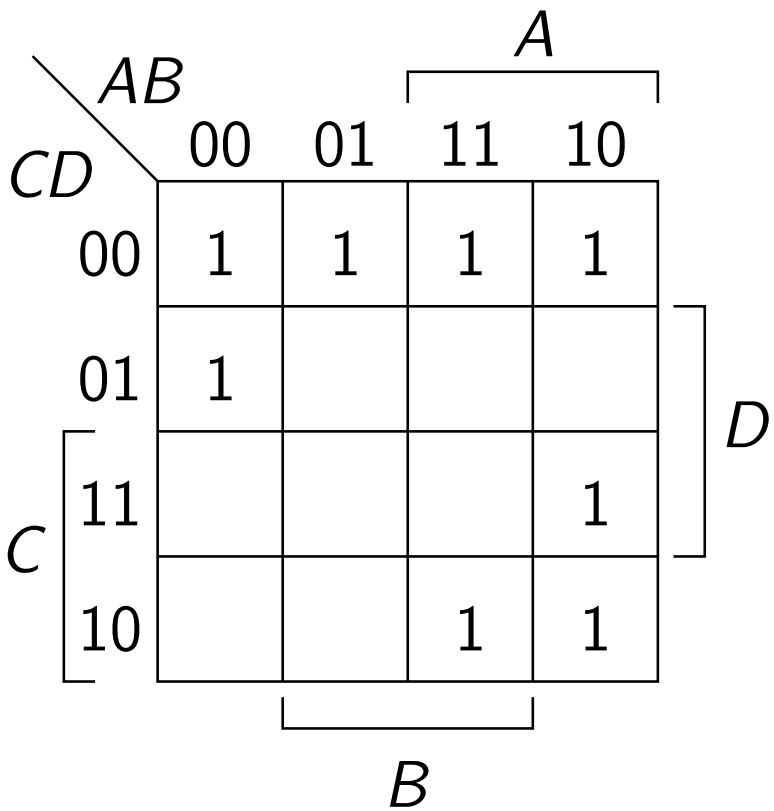
Example 1: Starting with a truth table

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Find a minimal POS expression for F.

Example 2: Starting with a K-map for F

K-map for F ...



Find a minimal POS expression for F.

Don't cares in minimal POS expressions

Fact: Given a truth table for F with some don't-care outputs, the truth table for $\overline{F}$ has don't-care outputs in exactly the same rows.

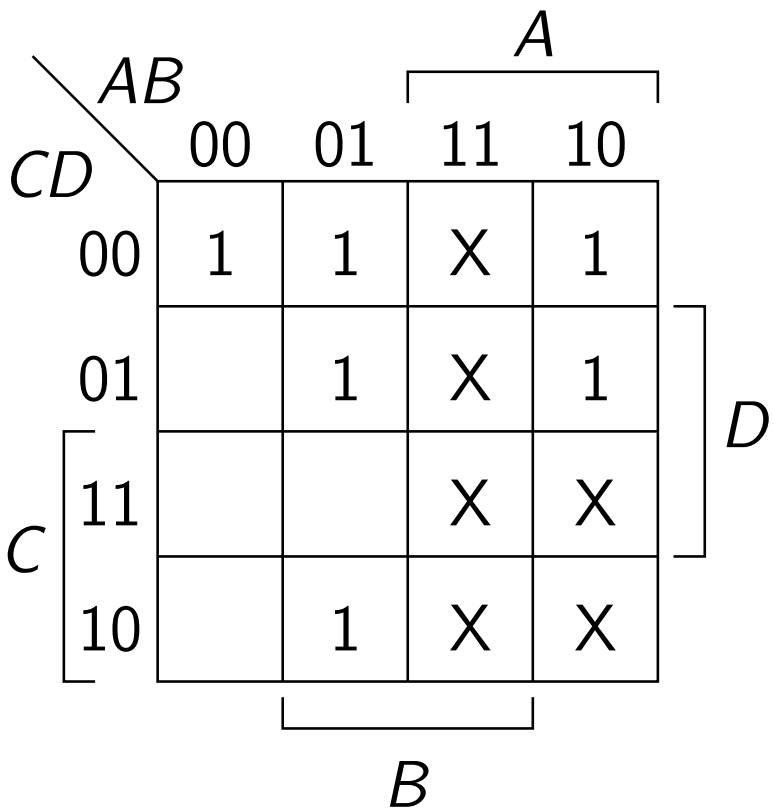
As a result, when going from a K-map for F to a K-map for $\overline{F}$ ...

- replace 1-cells with 0-cells;
- replace 0-cells with 1-cells;
- leave X-cells exactly as they appeared in the original map.

There's an example problem on the next slide ...

Example 3: K-map for F with don't-cares

K-map for F ...



Find a minimal POS expression for F.

Outline of Slide Set 5

K-maps with more than four variables

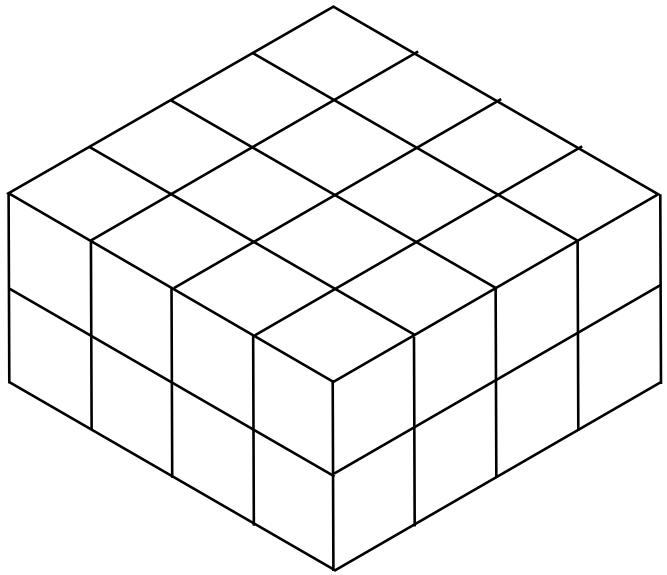
With three or four variables, adjacent cells are easy to spot, as long as you remember to look for groups that wrap around the edges of a map.

With five or six variables, it's more difficult to organize cells to make all adjacencies easy to discover.

There are a few different options for setting up 5- and 6-variable maps. We'll follow the method outlined in the ENEL 353 textbook used in Fall 2012 and earlier: Marcovitz A. B., *Introduction to Logic Design, 3rd ed.*, 2010, McGraw-Hill.

A 3-dimensional map for a 5-variable problem

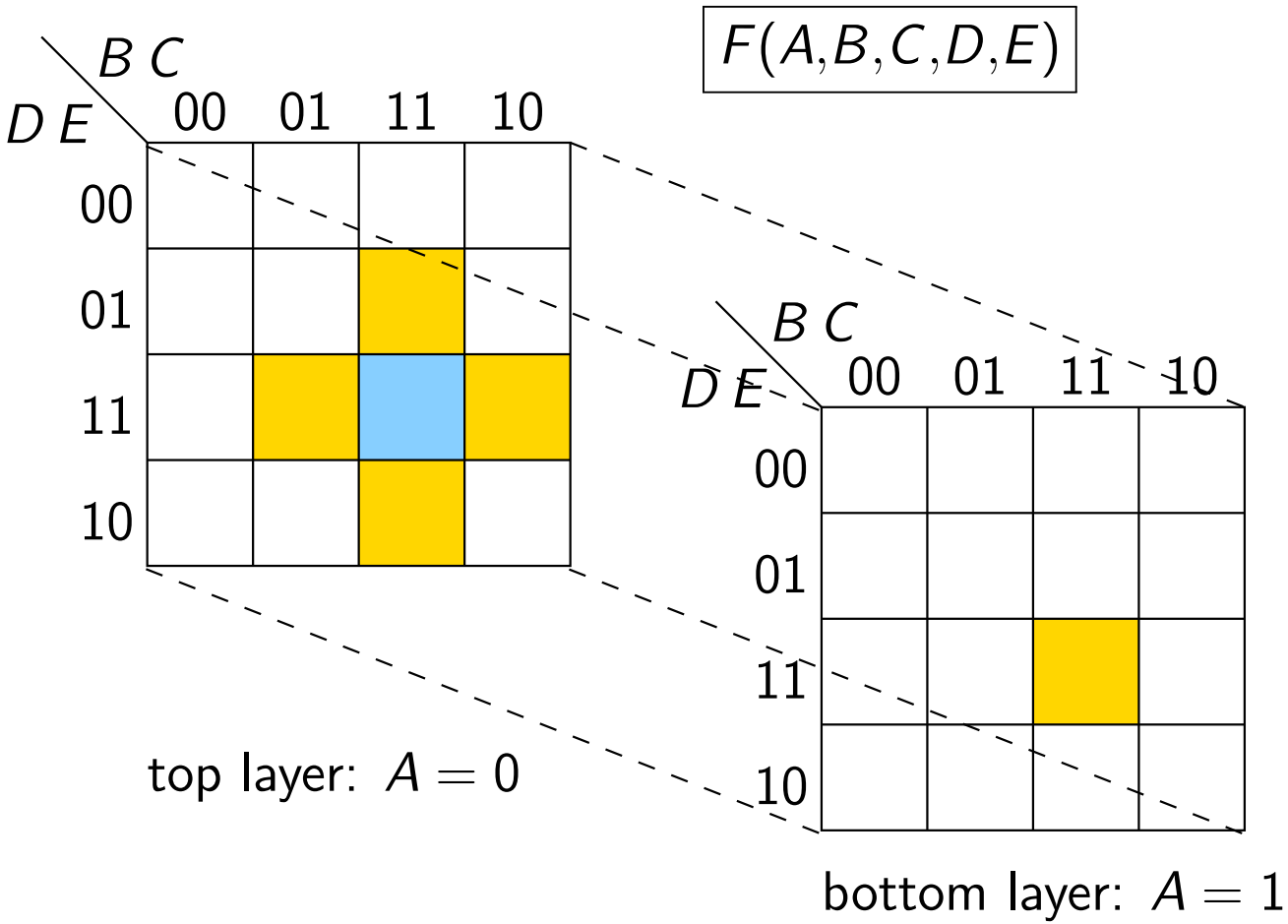
We can visualize the 32-cell map this way:



We look for **vertical** adjacencies as well as the **horizontal** adjacencies we're used to finding in 4-variable maps.

But marking and grouping 1-cells in a sketch like that is pretty much impossible . . .

2-dimensional representation of the 3-d map

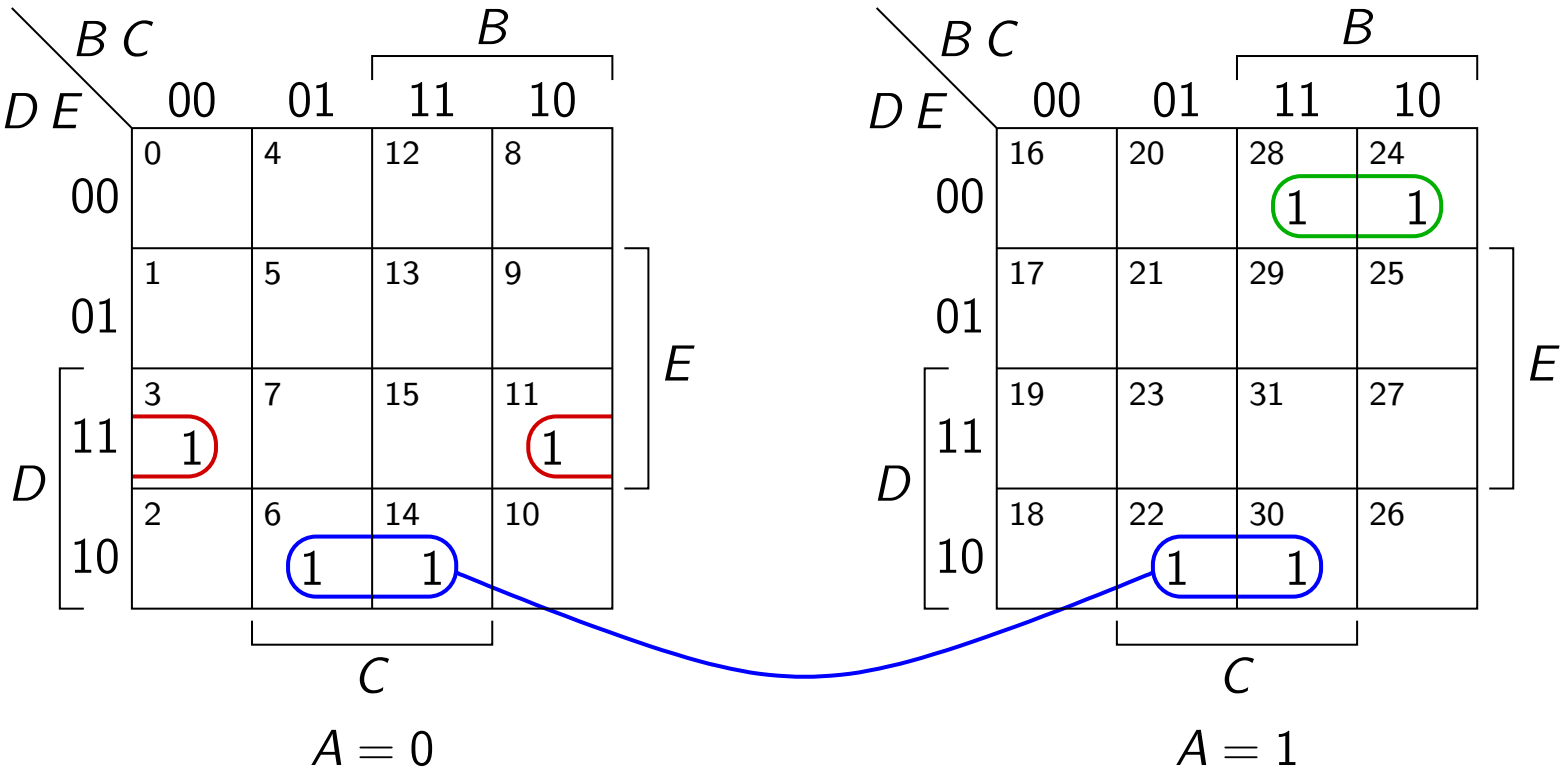


The five yellow cells are **all** adjacent to the blue cell.

Example 5-variable problem

Find a minimal SOP expression for
$F(A,B,C,D,E) = \Sigma(3, 6, 11, 14, 22, 24, 28, 30)$

Example 5-variable problem: Solution



$$F(A,B,C,D,E) = \bar{A}\bar{C}DE + AB\bar{D}\bar{E} + CD\bar{E}$$

6-variable K-maps

Visualize a $4 \times 4 \times 4$ cube of 64 cells.

Gray code ordering must be used in the vertical direction as well as the two horizontal directions.

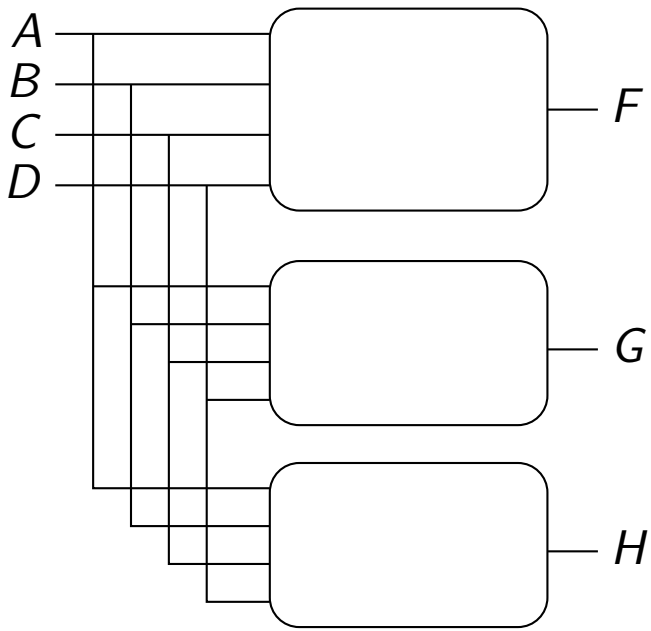
This year in ENEL 353 we will **not** do detailed examples of 6-variable K-maps, and you will **not** be tested on 6-variable K-maps.

Outline of Slide Set 5

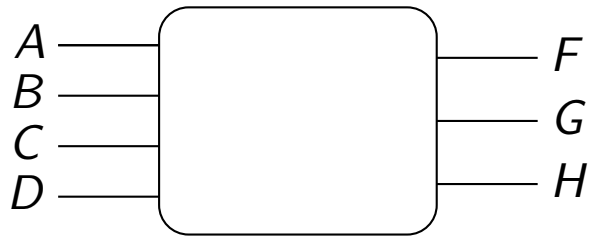
Multiple-output minimization problems

It's common to have combinational logic systems that have **multiple outputs** depending on the **same inputs**. It makes sense to look for ways to **share** logic gates between two or more functions . . .

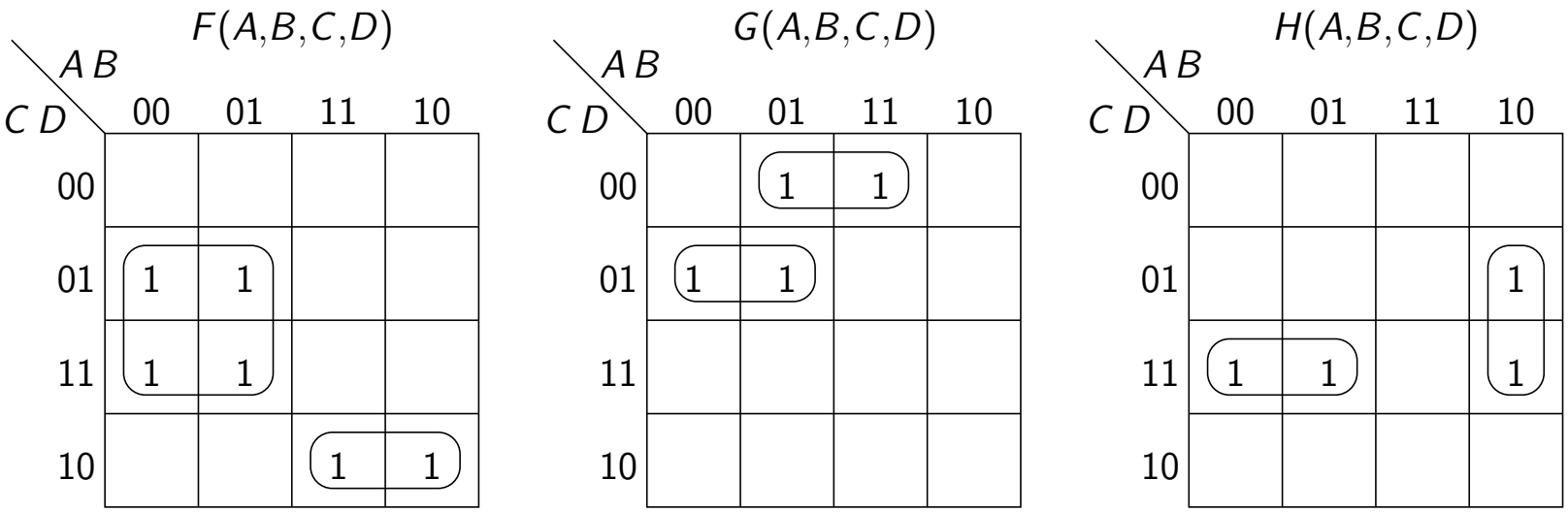
Separate circuits for F, G, H



Opportunity to share circuitry in producing F, G, H



An example 3-output problem: Here are K-maps for function F, G, and H, with common inputs A, B, C, and D . . .



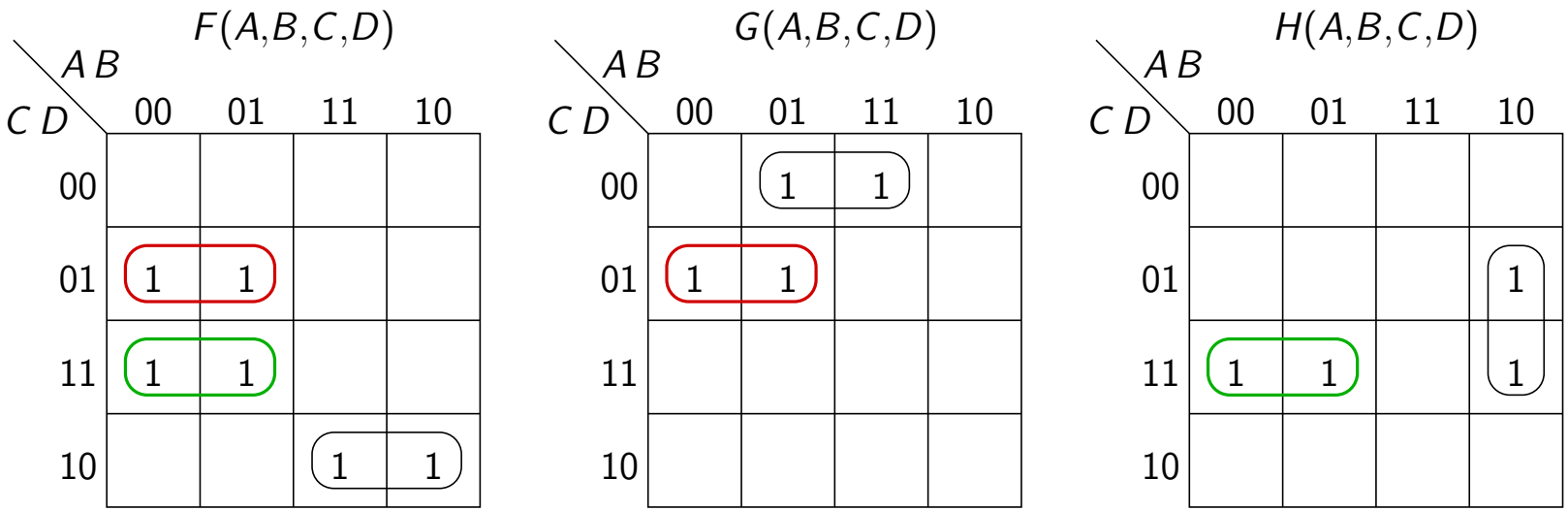
The groupings of 1-cells above show minimal SOP expressions for each of the outputs considered **independently of the other two** . . .

$$F = \bar{A}D + AC\bar{D} \qquad \text{2 AND gates, 1 OR gate}$$

$$G = \bar{A}\bar{C}D + B\bar{C}\bar{D} \qquad \text{2 AND gates, 1 OR gate}$$

$$H = \bar{A}CD + A\bar{B}D \qquad \text{2 AND gates, 1 OR gate}$$

By **splitting** one of the prime implicants of F we can **share** products that are already in use for G and H ...



The number of OR gates is 3, same as before, but the number of AND gates has been reduced to 5 from 6 ...

$$F = \bar{A}\bar{C}D + \bar{A}CD + AC\bar{D}$$
$$G = \bar{A}\bar{C}D + B\bar{C}\bar{D}$$
$$H = \bar{A}CD + A\bar{B}D$$

However, the OR gate that produces F now needs 3 inputs instead of 2.

Multiple-output problems: Remarks

The example shows that building minimal SOP expressions from prime implicants of individual outputs does **not** always minimize the overall number of products required to generate **all** the outputs with two-level SOP logic.

Finding minimum-cost circuits for multiple output circuits has been a topic in some past versions of ENEL 353, but in Fall 2019, we will **not** explore this topic in detail.