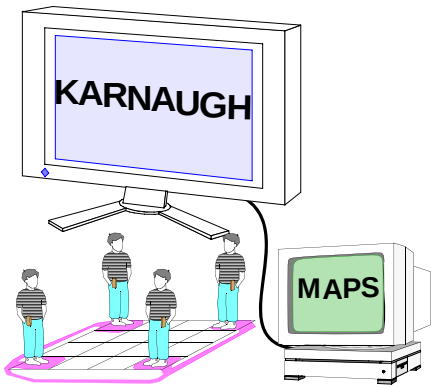
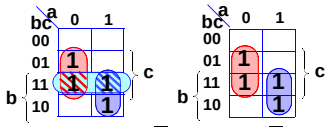
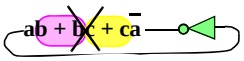
Digital Circuit Engineering



Consensus, use Karnaugh



$ab + bc + c\bar{a} = ab + c\bar{a}$



2nd Distributive

$(X + A)(X + B) = X + AB$

Simplification

$YX + X = X$

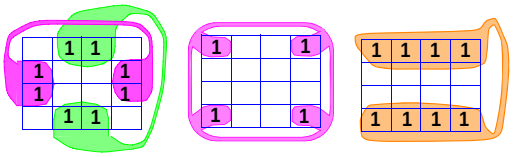
Absorption

$Y + X\bar{Y} = X + Y$

General DeMorgan

$\overline{F(a, b, \ldots z, +, \cdot, 1, 0)} \equiv F(\bar{a}, \bar{b}, \ldots \bar{z}, \cdot, +, 0, 1)$

Easily Forgotten Wrap-Arounds



Carleton University 2009

Chapter 3

Karnaugh Maps

Another form of the truth table

- Labelling Maps
- Circling Maps
- Minimizing Algebra by Maps
- Precautions when circling

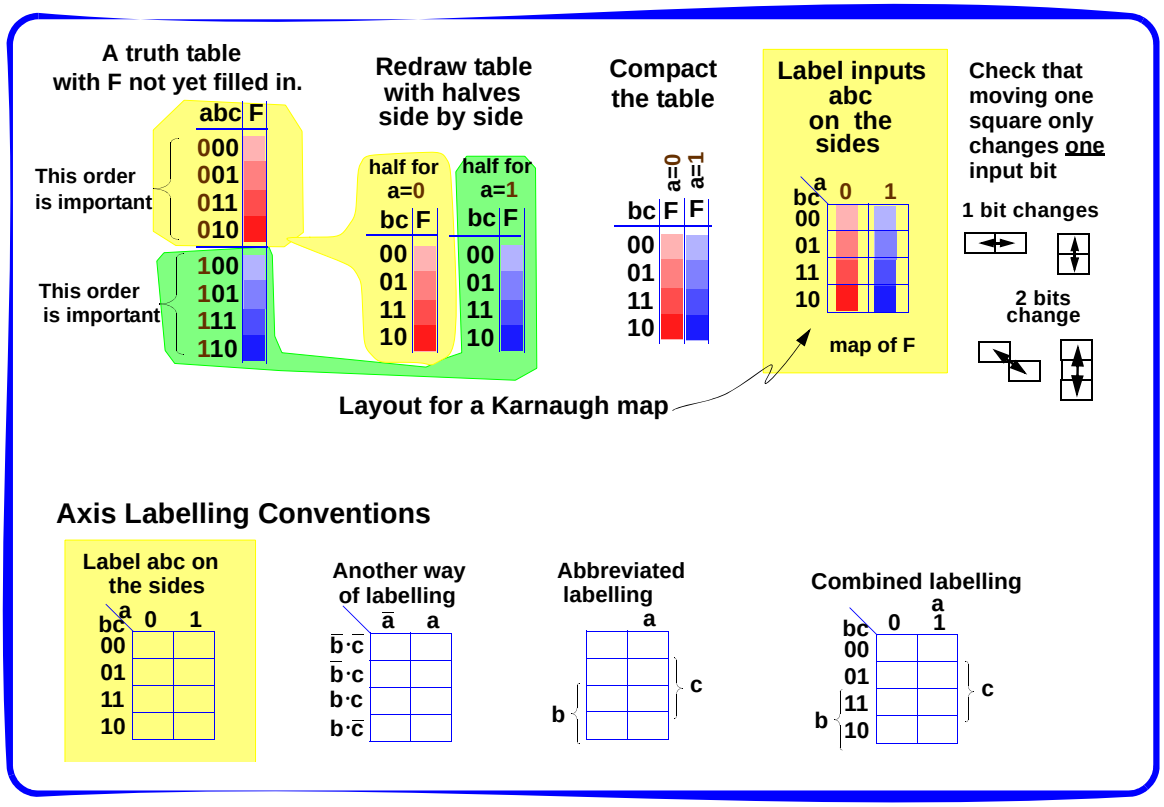
Don't Cares

- Where don't cares come from
- - Binary-Coded Decimal (BCD) digits
- Don't cares on Karnaugh Maps

Appendix

- XORs logic manipulation
- Using Karnaugh Maps with XORs

Karnaugh Maps; Equations From Truth Tables



Karnaugh Maps

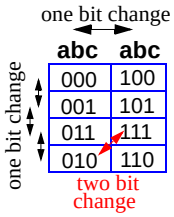
The map is like a truth table

Each square on the map represents a different input combination.
All possible input combinations are represented on the map.

The inputs are labelled around the edges of the map. Not inside the squares as shown on the right.

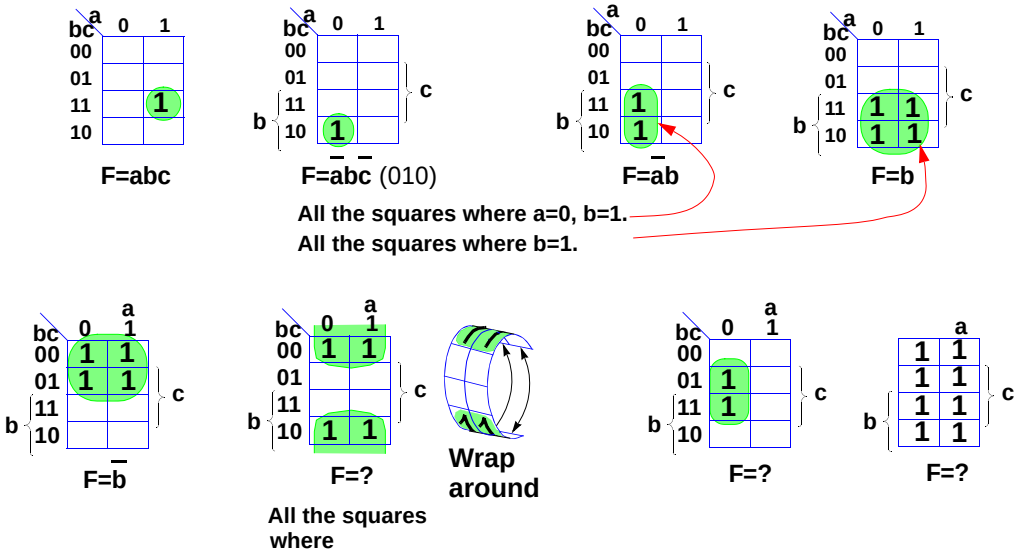
Arrangement of the squares

As one steps from one square to the next, either up, down, left or right, only one bit should change in a single step. If one goes to the nearest diagonal neighbour, two bits will change.



Karnaugh Maps; Representing AND Terms

Different Functions Using AND



Representing AND Terms

Any single square

(Top row, first two maps)

On these maps, any single square represents specific values for three variables, this is the same as a three term AND like $\bar{a}\bar{b}\bar{c}$

Any two adjacent squares

We made the maps so that only one variable changes at a time if one moves vertically or horizontally. (This is not true for diagonal movements). Thus two adjacent squares always have one common variable.

In the top row, third map, the squares $\bar{a}\bar{b}\bar{c}$ and $\bar{a}bc$ are 1. We can say $\bar{a}\bar{b}\bar{c} + \bar{a}bc = \bar{a}\bar{b}(\bar{c} + c) = \bar{a}\bar{b}$

This shows that any two adjacent squares can be represented by a two term AND.

Any three adjacent squares

Meaning less!, One can only loop if the number of squares is a power of 2.

Any four adjacent squares

(Top row, fourth map)

There are two ways to look at this. One is that all the squares where b=1 have a "1" in them, hence one can describe them as b.

Alternately one can note that squares that are "1" are $\bar{a}\bar{b}\bar{c} + \bar{a}bc + ab\bar{c} + abc = b(\bar{a}\cdot\bar{c} + \bar{a}\cdot c + a\cdot\bar{c} + a\cdot c) = b$.

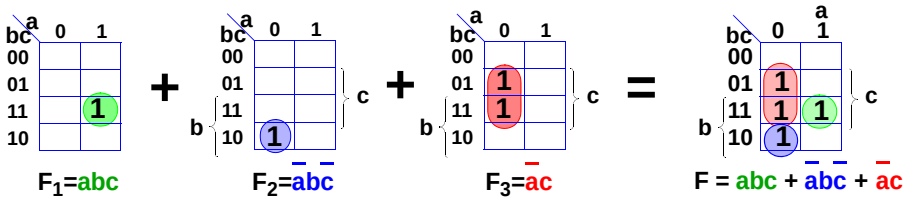
(bottom row, first two maps)
These represent $\bar{b}$ and $\bar{c}$ respectively.

Any eight adjacent squares

If all the squares on a map are "1", the function is F=1.

Karnaugh Maps; Joining AND Terms With ORs

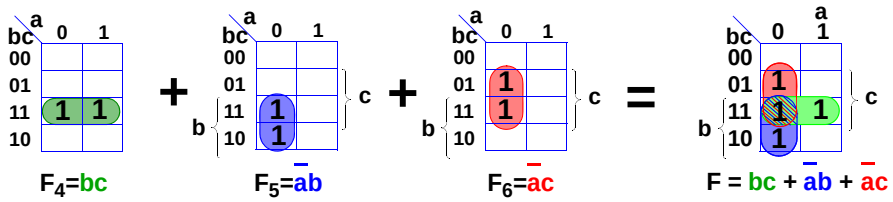
Joining $F = F_1 + F_2 + F_3$ on the map



F_1=abc F_2=$\bar{a}\bar{b}c$ F_3=$a\bar{c}$ F = abc + $\bar{a}\bar{b}c$ + $a\bar{c}$

OR together the AND terms, and place them on one map.

The terms can overlap.



F_4=bc F_5=$\bar{a}b$ F_6=$a\bar{c}$ F = bc + $\bar{a}b$ + $a\bar{c}$

Using the larger terms (F_4, F_5, and F_6) gives a smaller expression for F.
Bigger loops give smaller gates.

Circling a Map in Different Ways

Combining Maps

The OR of two maps, is a new map in which the squares are made 1 if there is a 1 in either (or both) of the initial maps

Same Map, Different Expressions

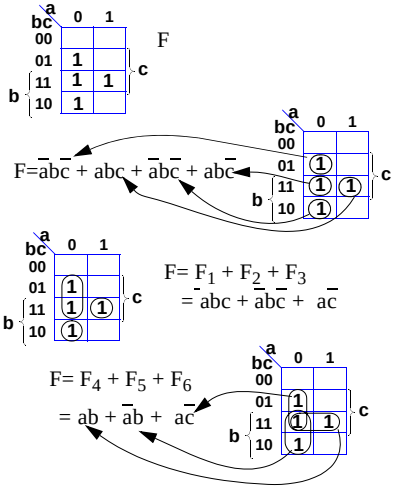
Take the truth table for a Boolean function F, written as a map.

One can loop it in several ways.

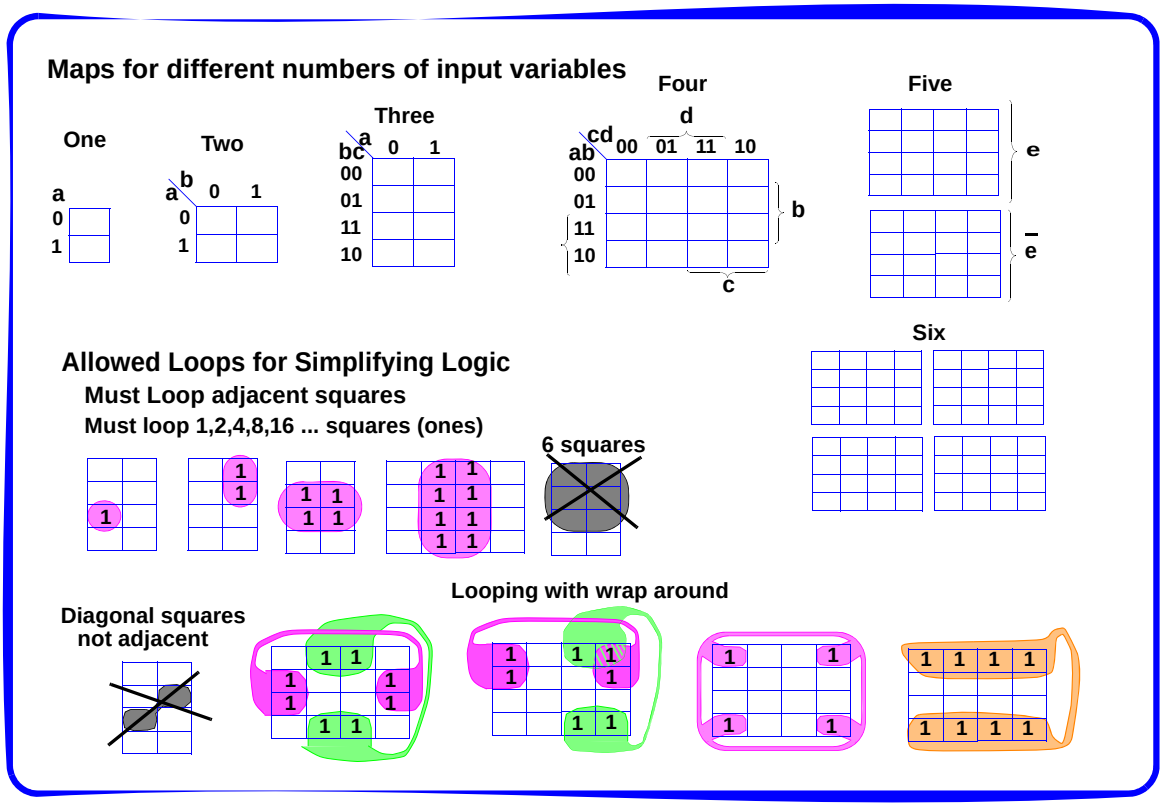
First one can loop individual "1"s. This gives a long expression for F.

Another way is to break it up as F_1, F_2 and F_3 as shown on the top line on the slide.

A third way is to break it into F_4, F_5, and F_6, as shown on the bottom line in the slide. This gives a smaller equation.



Maps for 1, 2, 3, 4, 5 and 6 Inputs; Legal Loops



Karnaugh Map Properties

Maps may have any number of variables, but-

- Most have 2, 3, 4, or 5 variable. One variable is simple, (2 squares).
- five variables has two 4x4 maps, one for when e=1, and one for when e=0.
- Six variables have 64 squares and were used in pre-computer days.
- Seven variables is past the limit of sanity. You will have 8 blocks of 16 squares each.

Rules for looping "1"s

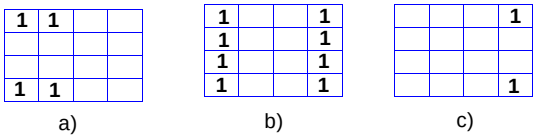
- "1"s on the maps can be looped to simplify logic.
- Only adjacent squares can be looped.
  Diagonally adjacent squares are not considered adjacent.
- loops must surround 1, 2, 4, 8, 16 ... squares. Not 3,5,6,7,9,10, ...
- The maps wrap around.
  A square on an edge is adjacent to the square on the opposite edge in the same column (row).

Larger loops give simpler logic, but-

- One must obey the above rules.
- There are exceptions, particularly with multi-output maps.

3-1. PROBLEM

loop the "1"s on the three maps shown.

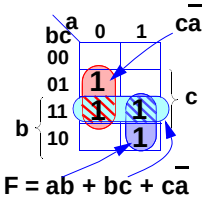


a) b) c)

Karnaugh Maps; Simplifying Equations

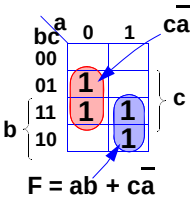
Looping to Minimize the Logic Expression

Simplify ab + bc + $c\bar{a}$

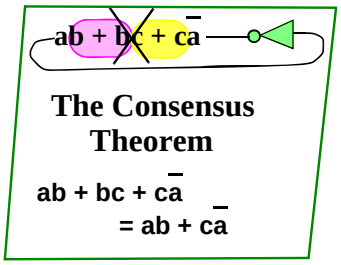


F = ab + bc + $c\bar{a}$

The bc term is redundant



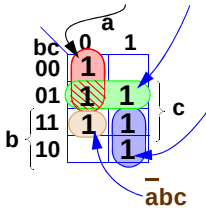
F = ab + $c\bar{a}$



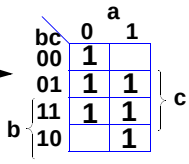
OR together the terms, and place them on one map.

Simplify F = $\bar{a}\cdot\bar{b}$ + $\bar{b}c$ + $\bar{a}bc$ + ab

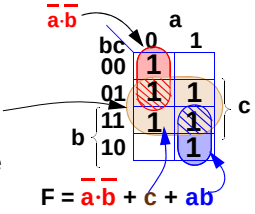
= $\bar{a}\cdot\bar{b}$ + $\bar{b}c$ + $\bar{a}bc$ + ab



These are the "1s" to loop



Use a bigger loop in the middle



Using the larger terms (loops) gives a smaller expression for F.

Simplification of Boolean Function

A Boolean function can be defined in many ways

- A truth table
- A circuit diagram.
- A Karnaugh map without loops
- A looped Karnaugh map.
- A Σ of Π expression.[1]
- A Π of Σ expression.
- A binary decision diagram (BDD).
- etc.

Any function has only one truth table and only one unlooped map.

There are usually several ways of circling the map or writing the algebraic expression for the same function.

One tries to find the best definition for some objective.

Possible Objectives

Smaller circuitry.
Lower powered circuitry.
Faster circuitry.

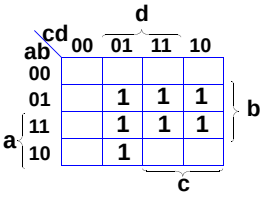
Making the circuit smaller is usually a good start toward making the circuit faster and lower powered.

1. Σ of Π , Π of Σ, and BDD are defined in a later chapter, or try the glossary in Moodle.

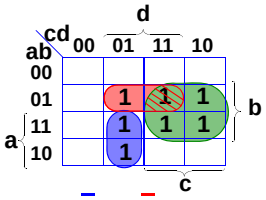
Karnaugh Maps; Best Loops to Reduce Logic

Simplification With 4-Input Maps

Simplify the function defined by this map

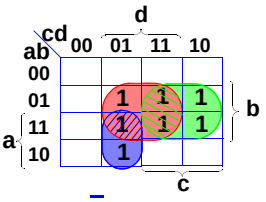


John's Solution



$F = \bar{a}c\cdot d + \bar{a}\cdot bd + bc$

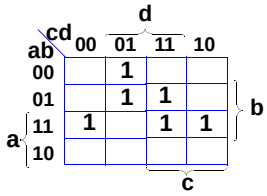
Tom's Solution



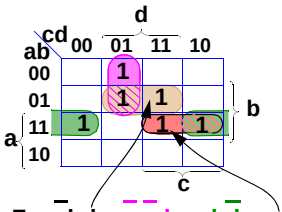
$F = \bar{a}c\cdot d + bd + bc$

Usually bigger is better for loops

Simplify the function defined by this map

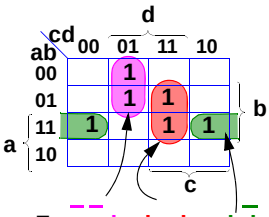


John's Solution



$F = \bar{a}bd + \bar{a}\cdot cd + ab\bar{d} + abc$

Tom's Solution



$F = \bar{a}\cdot cd + bcd + ab\bar{d}$

Simplification

Some Things to Do

Use the largest loop possible

John used $\bar{a}bd$ when he should have used bd.

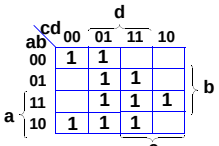
Check that overlap reduces, rather than increases logic.

John has two overlapping loops. Tom avoided both and saved a loop.

3-2. PROBLEM

Find the simplest expression for the logic function F[1] defined by looping the Karnaugh map on the right.

Get F with 9 letters. If it takes more, do Prob 3-1.a.

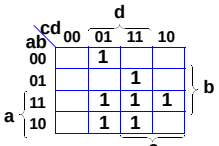


Map of F

3-3. PROBLEM

Find the simplest expression for the logic function F defined by looping the Karnaugh map on the right.

Get F with 11 letters.



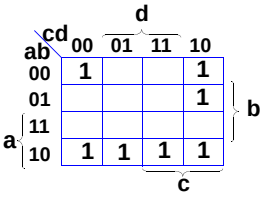
Map of F

[1] This logic function, written by ANDing together letters into terms, And ORing the terms, is called a *sum-of-products*

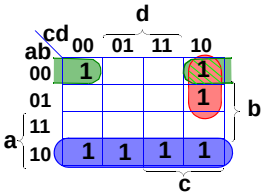
Karnaugh Maps; Best Looping to Reduce Logic

4-Input Maps, Looping Four Corners

Simplify the function defined by this map

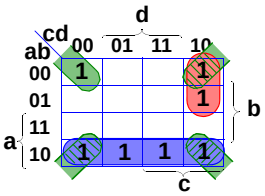


John's Solution



$F = a\cdot\overline{b} + \overline{a}\,\overline{c}\,\overline{d} + a\cdot\overline{b}\cdot\overline{d}$

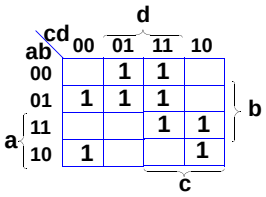
Tom's Solution



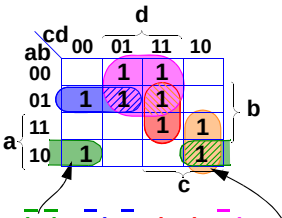
$F = \overline{ab} + \overline{a}\,\overline{cd} + \overline{b}\cdot\overline{d}$

Don't forget 4 corners

Simplify the function defined by this map

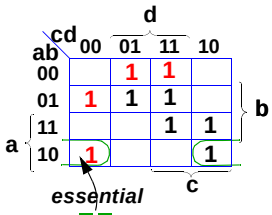


John's Solution



$F = a\cdot\overline{b}\cdot\overline{d} + a\cdot\overline{b}\cdot\overline{c} + bcd + \overline{a}\,\overline{d} + abc$

Your Solution



$F = a\cdot\overline{b}\cdot\overline{d} +$

Simplification Theory

Essential Terms[1]

A term (letters ANDed together that can be expressed by one loop) is essential if it contains at least one "1" that cannot be looped by any other loop, of the same or larger size.

"Your solution,"

The term $a\cdot\overline{b}\cdot\overline{d}$, is *essential* in that no loop (except smaller ones) will cover the square abcd=1000.

$\overline{a}\cdot b\cdot\overline{c}$, is *essential* in that no other loop (except smaller ones) will cover abcd=0100.

$\overline{a}\,\overline{d}$, is *essential* in that no other loop (except smaller ones) will cover abcd=0001 and 0011.

Any logic function derived by looping a map must contain these *essential* terms. There is no choice, so loop them first.

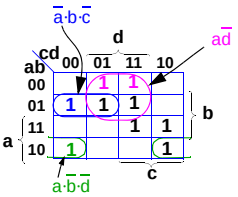
Squares Not Covered by Essential Terms

All the "1" squares except 1111 and 1110 are covered by the essential terms. One has a choice for terms to cover these squares.

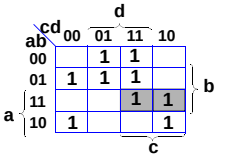
There are three terms that cover one or both of these squares. Choose the ones to give the simplest expressions.

3-4. PROBLEM

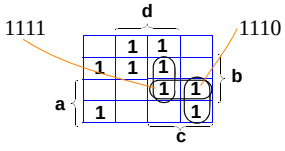
Find the essential terms for each of the 4 colored "1"s on map d). Then find the simplest logic expression found by looping map d).



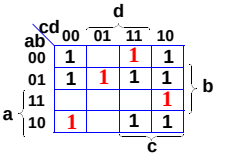
a)



b)



c)



d)

[1] The more rigorous definition is *essential prime product term,* or *essential prime implicant.* Prime means the loop is maximum size.

Example of How One Gets Don't Care Outputs

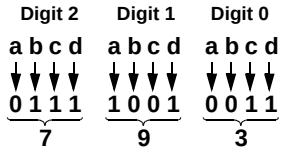
Binary Coded Decimals (BCD)

decimal digit	Binary representation abcd
0	0000
1	0001
2	0010
3	0011
4	0100

decimal digit	Binary representation abcd
5	0101
6	0110
7	0111
8	1000
9	1001

not used	Binary representation abcd
x	1010
x	1011
x	1100
x	1101
x	1110
x	1111

Representing a 3-digit number in decimal with 12 bits.



Map showing the values of abcd for each square

ab \ cd	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

Map showing the decimal equivalent of the input bits

a \ cd	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	x	x	x	x
10	8	9	x	x

d, b, c

"x" squares can never happen for BCD digits.

Binary-Coded Decimals

These are used mainly for sending numbers to displays which people have to read.

Many years ago they were used to do commercial arithmetic. The story was that converting decimal fractions to binary caused small errors which could accumulate and throw off your bank account.

For example $0.70 (decimal) = $0.1110,0110,0110,0110,0110,0110,0110,0110, ... (binary)

Binary-coded decimal digits use 4 bits.

The table above shows that six of the sixteen 4-bit combinations are unused.
If one has a circuit which has binary-coded decimal inputs there will be six input combinations which never happen. These are marked with "x".

If they never happen, then one does not have to worry about the circuits output for these input combinations.

These combinations are called *don't care* inputs and can be used to simplify the circuit.

3-5. PROBLEM

Write the year in binary coded decimal. In 2009, you should have 16 bits.

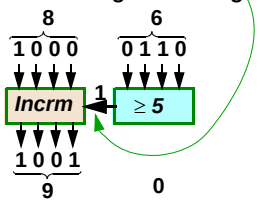
Don’t Care Outputs; Where They Come From

Rounding BCD numbers

Round 2-digit BCD numbers to 1-digit.

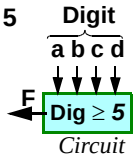
83 round to 80
86 round to 90
85 round to 90 (arbitrary choice)

If least sig digit ≥ 5
- Send increment sig to next dig



Design Circuit

Detect if a BCD digit ≥ 5



ab \ cd	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	x	x	x	x
10	8	9	x	x

Map locations of BCD digits

ab \ cd	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	x	x	x	x
10	1	1	x	x

F=digit ≥ 5

red “1”s show digits ≥ 5

Rounding BCD numbers.

There are three parts to this problem:

(1) Design a circuit to check if a BCD digit ≥ 5.

(2) Modify an adder circuit to increment a BCD digit.

(3) Put it all together.

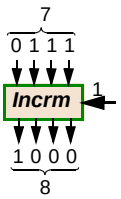
We will only do part (1) in the slide above. Part 2 is a problem below.

Rounding Using the Don’t Cares

3-6. PROBLEM (Fairly difficult)

a) Design the part (2) of the rounding circuit. A circuit that increment a BCD digit
- One only needs half-adders, not full adders.
- there is a carry in from the rounding signal.

If the digit coming into the incrementer was 9, incrementing it will overflow. For simplicity let
1001 + 1 = 1010



b) A real BCD incrementer would increment 9 to give 0 with an overflow output signals. Add a circuit to check for a 1001 and an increment input, and give a 0000 plus an overflow.

Karnaugh Maps: Using Don't Cares

Don't Cares In Karnaugh Maps

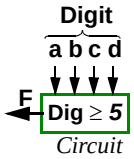
Detect if a BCD digit is 5 or more.

ab \ cd	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	x	x	x	x
10	8	9	x	x

Map locations of BCD digits

ab \ cd	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	x	x	x	x
10	1	1	x	x

F= digit ≥ 5



Use of "Don't Cares".

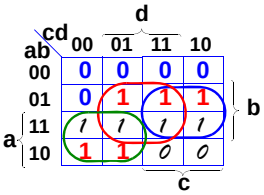
The BCD digits > 9 never happen
We don't care about output for them.
Make these outputs "*d*" on the map.

"*d*" may be looped or not as desired.
Loop to minimize logic
Here we looped 4 out of 6 "*d*"s, *to get*

F= $\bar{a}c$ + db + bc

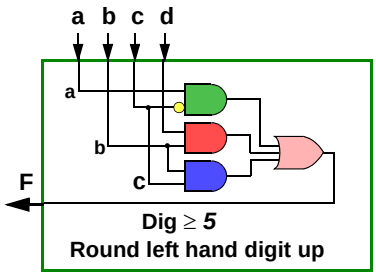
ab \ cd	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	*d*	*d*	*d*	*d*
10	1	1	*d*	*d*

What's with this?
See notes below



Inputs to cause the six "*d*" outputs never happens, but if they did, with the this looping:

- the 4 looped outputs would then be "1", and
- the 2 unlooped ones would the be "0".



Don't Cares In Karnaugh Maps

If one has input combinations that never happen, one does not care what outputs they generate because those outputs can never happen.

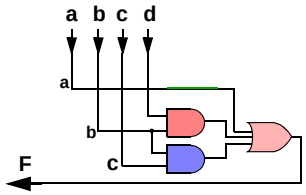
We put don't cares on the map squares for input combinations that never happen.

One can loop these don't cares or not as convenient.

There is a common error on the slide.
The loop **$\bar{a}c$** could be extended to include all of **a**.

This would give the final equation as

F= a + bd + bc



3-7. PROBLEM (no *d*s)

Take the hex digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Plot their location on a Karnaugh map in the same way the BCD digits were plotted.

Then design a logic circuit which will use four bits w,x,y,z (defining a hex digit as input, and give a high output if the digit is divisible by 3, i.e. it is 3, 6, 9, C or F.

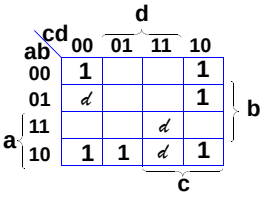
3-8. SUPPLEMENTAL PROBLEM (with *d*s)

Design a circuit which will use four bits a,b,c,d defining a BCD digit as input, and gives a high output if the digit is divisible by 3. Utilize the inputs that cannot happen, to give don't care outputs, and hence simplify the logic.

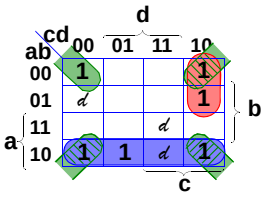
Karnaugh Maps; Examples With Don't Cares

Simplification With Don't Cares

Simplify the function defined by this map



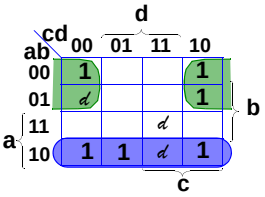
John's Solution



$F = \bar{a}\bar{b} + \bar{a}\bar{c}\bar{d} + \bar{b}\cdot\bar{d}$

Don't have to loop all the "d".

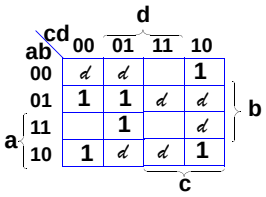
Tom's Solution



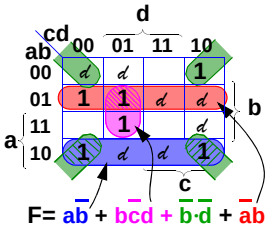
$F = a\cdot\bar{b} + \bar{a}\cdot\bar{d}$

4 corners not so good

Simplify the function defined by this map

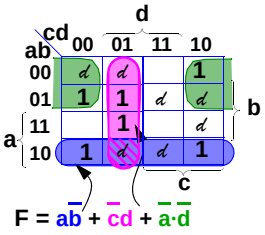


John's Solution



$F = \bar{a}\bar{b} + bcd + \bar{b}\cdot\bar{d} + \bar{a}\bar{b}$

Tom's Solution



$F = \bar{a}\bar{b} + \bar{c}\bar{d} + \bar{a}\cdot\bar{d}$

3-9. PROBLEM: K-MAP WITH DON'T CARES

One way to design a comparator for two binary numbers is shown. It is made of blocks, each of which compares two bits. The example compares two, 3-bit, binary numbers $X=x_2x_1x_0$ and $Y=y_2y_1y_0$. It uses three blocks.

Each block compares inputs x_i with y_i with the inputs A_i, B_i from the comparison done for higher order bits.

The result give outputs A_{i-1} and B_{i-1}.

Thus if A_i=1, coming into a block, then the left hand bits have shown $x_2x_1x_0 > y_2y_1y_0$ without even bothering to check x_i and y_i.

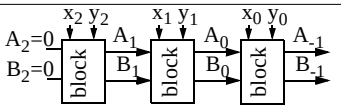
Similarly any if B_i=1 then $y_2y_1y_0 > x_2x_1x_0$.

A typical block is shown. We drop the cumbersome subscripts and write A^-, B^- to tell the A, B inputs from the outputs.

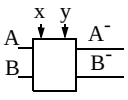
The "- -" in the 5th line of the truth table inputs means:
If A,B = 1,0 then, no matter what x and y are, the output is $A^-,B^- = 10$.

Do not confuse these with the don't cares in the outputs, "d", which are the result of input combinations that never happen.

Complete the Karnaugh map for A^- including the don't cares. Then deduce the expression for A^- which should have 4 letters.

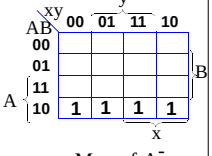


If A=10, then the left hand bits have already shown $x_2x_1x_0 > y_2y_1y_0$ independent of x and y Thus the block should send out A^-=1. Since one cannot have both numbers larger, B^-=0.

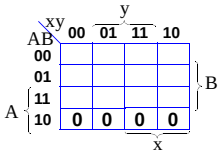


A^- =1 if x>y or AB=10
B^- =1 if x<y or AB=01
AB=11 never happens

A B x y	A^- B^-
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	0 0
1 0 - -	1 0
0 1 - -	0 1
1 1 - -	d d



Map of A^-



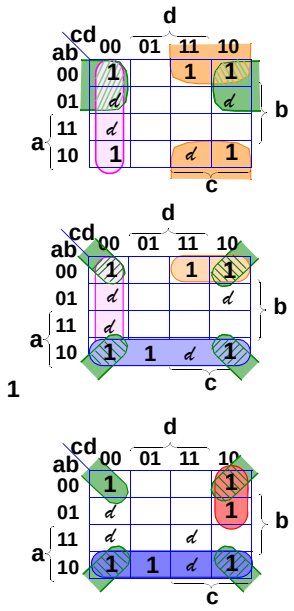
Map of B^-

Maps With Don't Cares; Common Mistakes

Don't Cares

Common mistakes

1. **Looping don't cares when there is no need. Remember you loop them only if convenient.** *(top)*
2. **Over looping.** *(mid)*
3. **Forgetting wrap-around.** *(mid, bot)*
4. **Not enclosing don't cares which would make the loops larger.** *(mid, bot)*



Common Mistakes with Don't Cares

Top: The four corners would be better. There is no need to include the upper two "*d*"s.

Middle: The four corners are redundant. Also the top **orange** $\bar{a}\cdot\bar{b}c$ oval could be doubled with wrap around.

Bottom: The red oval $\bar{a}c\bar{d}$ could be wrapped around to cover four squares.

Appendix I: Karnaugh Maps With XORs

Using XORs

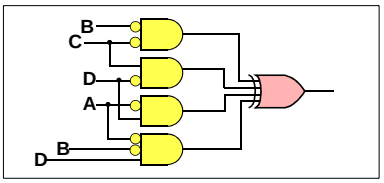
XOR Sum of Products (XSofP)

Also called *Exclusive SofP* or *Extended SofP*

XORs are used instead of the ORs used in SofP

Example of XSofP:

$\overline{B}\,\overline{C} \oplus C\overline{D} \oplus \overline{A}D \oplus \overline{A}\,\overline{B}D$



In CMOS (and most other) logic families
XOR gates are larger and slower than OR/NOR.
Thus circuits are not conceptually designed using XORs.

However:

- **For some very-fast logic types (*not CMOS*) XORs cost less than NORs.**
- **Some circuits are easier to conceptually design thinking about XORs.**
  like: adders
  error-checking circuits
  Gray-code to binary converters.
- **Quantum and DNA logic are designed with XORs**
  These logic types are not yet practical but soon?

Using XORS

For some very fast logic types, XORs are more advantageous.
Very fast logic, is often differential (supply both X and $\overline{X}$).
Differential gates supply X and $\overline{X}$ at no additional cost.
With both X and $\overline{X}$ available an XOR can be made for under twice the cost of a NAND.
Two such fast differential logic type are:
MOS Complimentary-Mode Logic (MCML) and
Emitter-Coupled Logic (ECL).

Quantum[1,2,3] and DNA logic[4]
These logic technologies naturally tend to use XOR rather than NAND or NOR.

Smith[5] gives a more complete but simple review of XOR XNOR circuits.

Sum-of-Products and Extended (or Exclusive) Sum-of-Products

The *sum-of-products* form of a function is a strict OR of ANDs way of writing a function, there must be no brackets or long overbars; however single inverting bars are allowed. See the notes on factoring.

Examples:
$abc + bcd + d + \overline{a}cde$ $\quad b\overline{c}d + d$ $\quad b + d + e$ $\quad d$

Not sum-of-product
~~$abc + bcd(d + \overline{a}cde)$~~ ~~$abc + \overline{bcd} + \overline{a}cde$~~

The *extended sum-of-product* (X-SofP) or (X-ΣofΠ) is a sum-of-product in which XOR is used instead of OR

Examples: $abd \oplus acd \oplus e \oplus \overline{a}cde$ $\quad a\overline{c}d \oplus d$ $\quad b \oplus d \oplus e$ $\quad e$

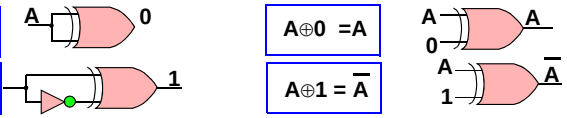
1. Prof. Marek Perkowski, *Reversable Logic Synthesis . . .*, http://web.cecs.pdx.edu/~mperkows/=PUBLICATIONS/PDF-2003/RM03.pdf
2. Riley Perry, *The Temple of Quantum Computing*, http://www.toqc.com/TOQCv1_1.pdf; corrections in http://www.toqc.com/
3. Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information*, Cambridge Univ Press, 2000.
4. A.Okamoto,K. Tanaka, and I. Saito,*DNA Logic Gates*, http://bi.snu.ac.kr/Courses/g-ai04_2/DNA%20Logic%20Gates_namjw.ppt
5. Prof. D.W. Smith, *Digital Logic Systems*, http://users.senet.com.au/~dwsmith/boolean2.htm

Commonly Used XOR Formulas

Rules using XOR

Basic

$A \oplus A = 0$ $\quad A \oplus \overline{A} = 1$ $\quad A \oplus 0 = A$ $\quad A \oplus 1 = \overline{A}$



Inversion of XOR

$\overline{A \oplus B} = \overline{A} \oplus B = A \oplus \overline{B}$ *(I x)*

Commutative

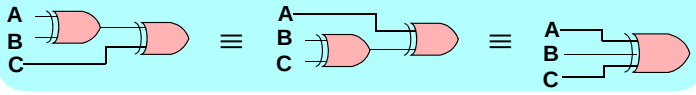
$X \oplus Y = Y \oplus X$

No brackets needed

Thus one can write $A \oplus B \oplus C$

Associative

$A \oplus (B \oplus C) = (A \oplus B) \oplus C$



Distributive (D1 equivalent)

$A(B \oplus C) = AB \oplus AC$ *(Dx1)*

No D2 Equivalent (No Dual for *(Dx1)*)

There is no *(Dx2)* ~~$(AB) \oplus C) = (A \oplus B) \bullet (A \oplus C)$~~

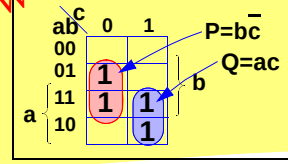
Disjunctive Theorem *(Dj)*

When $P \cdot Q = 0$, On a map this means the loops for P and Q is don't overlap.

Then:

$P + Q = P \oplus Q$ *(if $P \cdot Q = 0$)*

NEW



$F = P + Q$

$F = b\overline{c} + ac$

loops don't overlap

Hence $P \cdot Q = 0$

$F = P \oplus Q = (b\overline{c}) \oplus (ac)$

XOR Formula Details

$A \oplus A = 0$ $\quad A \oplus \overline{A} = 1$ $\quad A \oplus 0 = A$ $\quad A \oplus 1 = \overline{A}$

The way to remember. Substitute

A	B	B⊕A	B=A A⊕A	B=$\overline{A}$ A⊕$\overline{A}$	B=0 A⊕0	B=1 A⊕1
0	0	0	0		0	
0	1	1		1		1
1	1	0	0			0
1	0	1		1	1	

- An odd number of inversion bars inverts an XOR sequence.

$\overline{A \oplus B \oplus C \oplus D} = A \oplus \overline{B} \oplus C \oplus D = \overline{A} \oplus B \oplus \overline{C} \oplus \overline{D} = A \oplus (\overline{B \oplus C}) \oplus D$

- $A \oplus (B \oplus C) = (A \oplus B) \oplus C$ You can put the brackets anywhere, hence you don't need them.
- $A(B \oplus C) = (AB) \oplus (AC)$ D1 equivalent *(Dx1)*.

  There is no D2 (dual type) distributive law.

With XORs, brackets should **not** be assumed unless they don't make a difference.

Writing B⊕AB makes it easy to confuse (B⊕A)B with B⊕(AB)

Writing B⊕A⊕C is OK since (B⊕A)⊕C = B⊕(A⊕C)

Example:

1 $A + B = A \oplus B \oplus (AB)$

Proof:

if A=1=B *then*

lhs = A+B=1; rhs=1⊕1⊕1=1, using 1⊕1= 0, 0⊕1= 1

otherwise AB=0 *and*

lhs = A+B ; rhs= A⊕B⊕0 = A⊕B = A+B using X⊕0 =X disjunctive theorem, since AB=0

3-10. PROBLEM

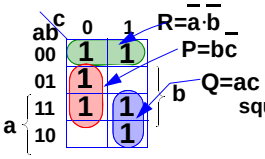
Look at Dx1 (XOR distributive law) and Dx2. Then deduce if Duality holds with ⊕ instead of +.

Basis for XK-maps; K-Map Parity

Disjunctive Theorem (cont.)

When P·Q· ... ·R =0 then

P + Q + ... R = P ⊕ Q ... ⊕ R



F = P + Q + R

loops don't overlap

Hence PQR = 0

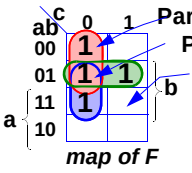
square-by-square AND of P·Q·R = 0

∴ F = P ⊕ Q ⊕ R

Extension of *(Di)* to Parity

When the odd parity of P,Q,R =1 on all squares of the map then

P+Q+R = P ⊕ Q ⊕ R



map of F

Parity=1 (1 loop covers square)

Parity=1 (3 loops cover square)

Parity=0 (no loops cover square)

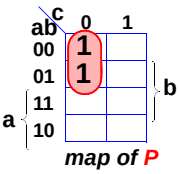
F = P ⊕ Q ⊕ R

F = a·c ⊕ bc ⊕ ab

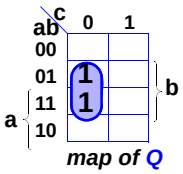
Explaination of example

When three or one (an odd number) loops overlap,

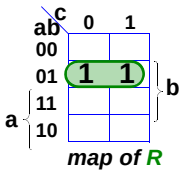
F=1



map of P

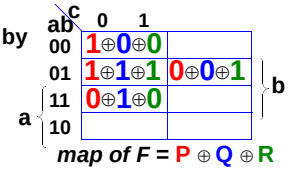


map of Q

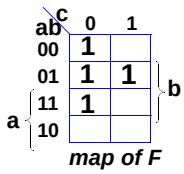


map of R

Square by square XOR



map of F = P ⊕ Q ⊕ R



map of F

Meaning of Parity

Parity of binary numbers.

Before odd parity meant the number of "1" bits in the number was an odd number. 100110 has odd parity 1

With maps, it means the number of loops around the square under consideration is an odd number.

XK-Map Examples

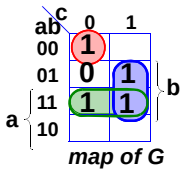
Disjunctive Theorem (cont.)

Loops that work for XOR are illegal for OR

Sometimes, with XOR, one can loop "0"s.

But one must have 2 loops, 4 loops ... overlapping

The loops must overlap with parity 0, and one must have the function = 0 on those squares.



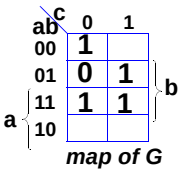
map of G

Using normal sum-of-products

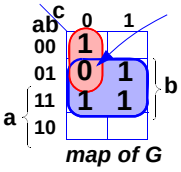
$G = \bar{a}\cdot b\cdot\bar{c} + bc + ab$

overlap on square

Parity=0 (2 loops cover square) but G=0 in this square

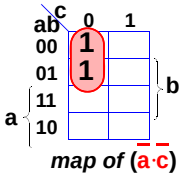


map of G

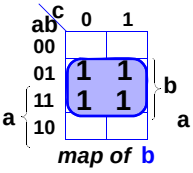


map of G

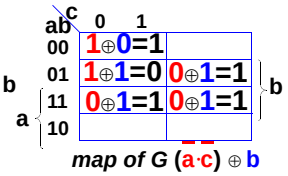
$G = (\bar{a}\cdot\bar{c}) \oplus b$



map of $(\bar{a}\cdot\bar{c})$



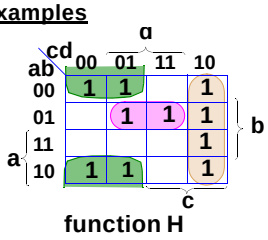
map of b



map of G $(\bar{a}\cdot\bar{c}) \oplus b$

XK-Map Examples

Three Examples



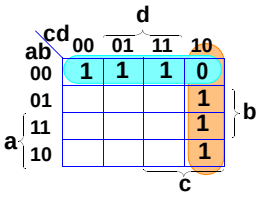
function H

$H = \overline{b}\,\overline{c} + \overline{a}bd + c\overline{d}$

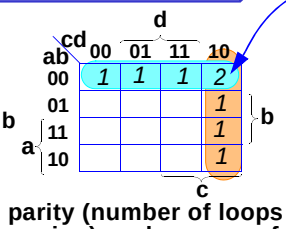
(Dj) tells us $P \oplus Q \oplus R = P+Q+R$ when PQR=0

here $P=\overline{b}\,\overline{c}$ $Q=\overline{a}bd$ $R=c\overline{d}$; PQR=0 on all squares

$\therefore\ H = \overline{b}\,\overline{c} \oplus \overline{a}bd \oplus c\overline{d}$



function G



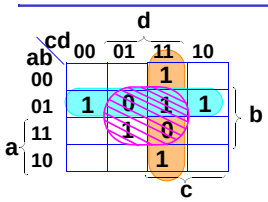
parity (number of loops covering) each square of G

$\overline{a}\,\overline{b} + c\overline{d}$ *equals* $\overline{a}\,\overline{b} \oplus c\overline{d}$ except on square $\overline{a}\,\overline{b}cd$, there:

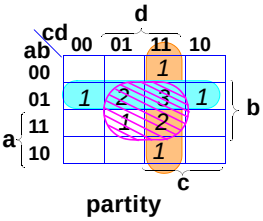
Parity $(\overline{a}\,\overline{b}, c\overline{d}) = (\overline{a}\,\overline{b}) \oplus (c\overline{d}) = 0$

Thus $\overline{a}\,\overline{b} \oplus c\overline{d} = 0$ *on* $\overline{a}\,\overline{b}cd$

$\Rightarrow G = \overline{a}\,\overline{b} \oplus c\overline{d}$



function K



partity

$K = \overline{a}b \oplus bd \oplus cd$

on squares covered by 1 terms the parity is 1 $\Rightarrow$ K=1

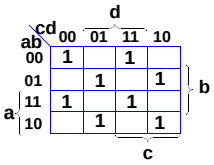
on squares covered by 2 terms the parity is 0 $\Rightarrow$ K=0

on squares covered by 3 terms the parity is 1 $\Rightarrow$ K=1

XOR is good for checker board patterns.

Look at function K on the bottom of Slide 17. Notice how simple the XSofP expression is.

In general, the more the map resemples a checker board pattern, the simpler the XSofP expression, and the more complex the SofP expression.



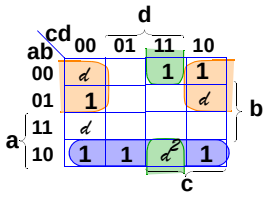
3-11. PROBLEM, XSOFP

a) Write the simplest S-of-P expression for the checkerboard pattern on the right.

b) Write the simplest XS-of-P expression for the same pattern.
(answer has 4 letters)

Three More Examples of X-SofP Maps

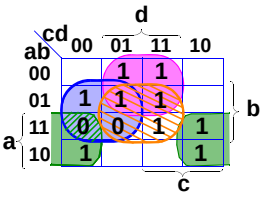
function L



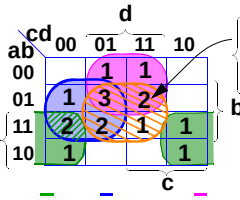
L= $\bar{a}\,\bar{d} \oplus \bar{b}cd \oplus a\bar{b}$

Don't cares are especially easy.
∅ squares can have any parity.

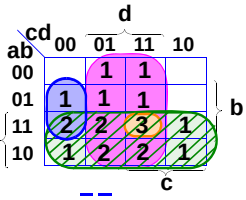
Showing parity (number of covers) of squares



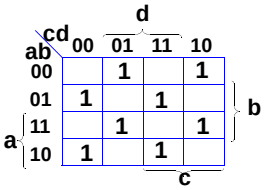
function M



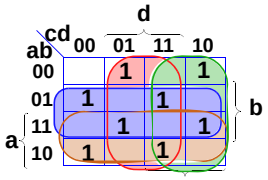
M = $a\cdot\bar{d} \oplus b\cdot\bar{c} \oplus bd \oplus \bar{a}d \oplus \bar{a}bcd$

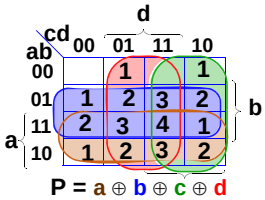


M = $a \oplus b\cdot\bar{c}\cdot\bar{d} \oplus d \oplus abcd$



function P





P = $a \oplus b \oplus c \oplus d$

2. Transposition theorem; a more advanced, but very powerful tool**.**

Given three functions of the same variables, f=f(a,b,c), g=g(a,b,c), and h=h(a,b,c).

And further f = g⊕h, then:

g = f⊕h and h = g⊕f

Proof:

Given f = g⊕h

⇒ f⊕h = g⊕h⊕h *XOR both sides with h*

= g *h⊕h = 0; g⊕0 = g*

qed.

Example: A Gray code to Binary code converter

The *Reflected Gray code*, $g_N,...g_2,g_1,g_0$, is derived from the binary code, $b_N,...b_2,b_1,b_0$, bit-by-bit using:

$g_k = b_k \oplus b_{k+1}$, **except** for the most significant bits where $b_N = g_N$.

To convert Gray code to binary code, use the transposition theorem to give:

$b_k = g_k \oplus b_{k+1}$. (k<N)

Thus to convert the four-bit Gray code, g_3,g_3,g_1,g_0, to b_3,b_2,b_1,b_0 use:

$b_3 = g_3$

$b_2 = g_2 \oplus b_3 = g_2 \oplus g_3$

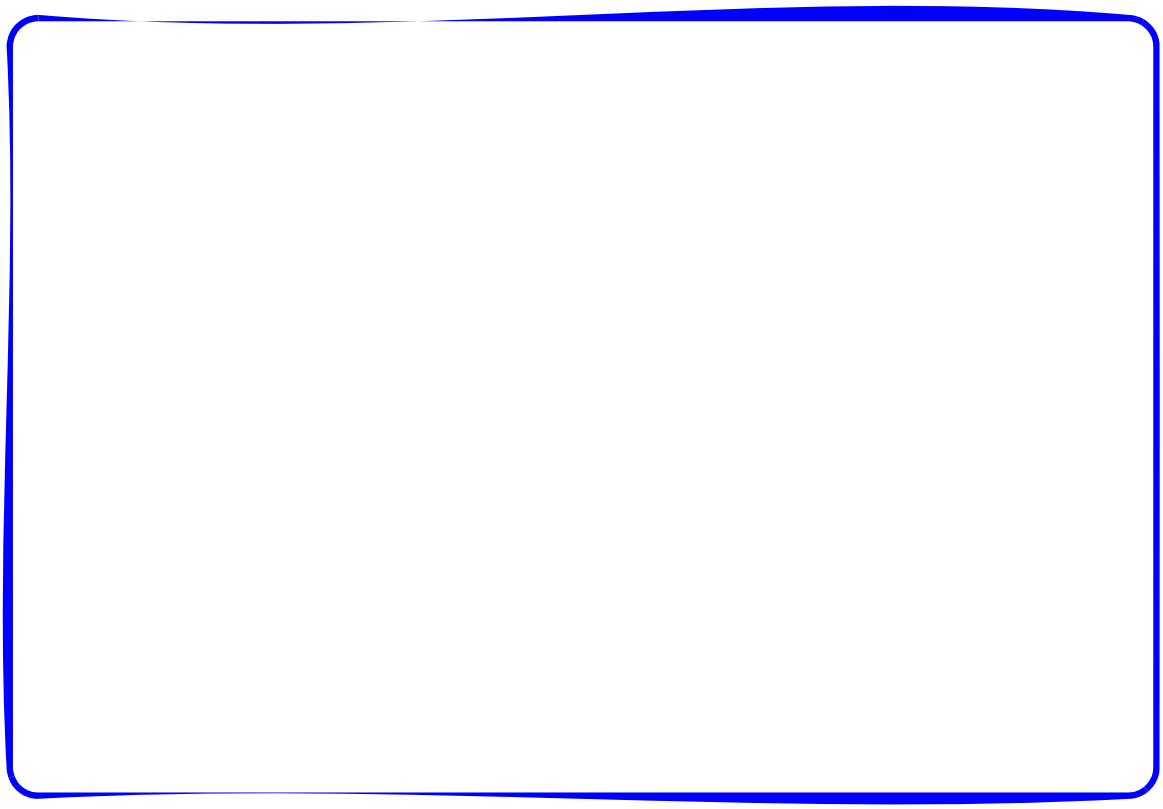
$b_1 = g_1 \oplus b_2 = g_1 \oplus g_2 \oplus g_3$

$b_0 = g_0 \oplus b_1 = g_0 \oplus g_1 \oplus g_2 \oplus g_3$

3-Bit Gray Code

Dec	$b_2b_1b_0$	$g_2g_1g_0$
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 1
3	0 1 1	0 1 0
4	1 0 0	1 1 0
5	1 0 1	1 1 1
6	1 1 0	1 0 1
7	1 1 1	1 0 0

When counting in Gray Code, only one bit changes at a time.



XOR Formula Details