Computer Architecture

Digital logic: Part 1

Boolean algebra and simplifying equations

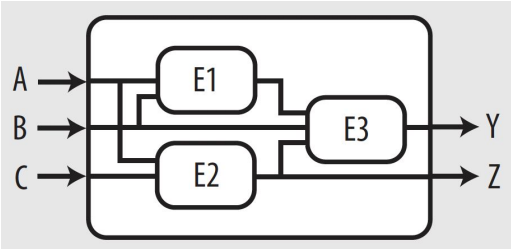
Circuit (in digital electronics)

A circuit can be viewed as a black box with

- one or more discrete-valued input terminals
- one or more discrete-valued output terminals
- a functional specification describing the relationship between input and output
- a timing specification describing the delay between inputs changing and outputs responding.

Inside the black box, it is composed of

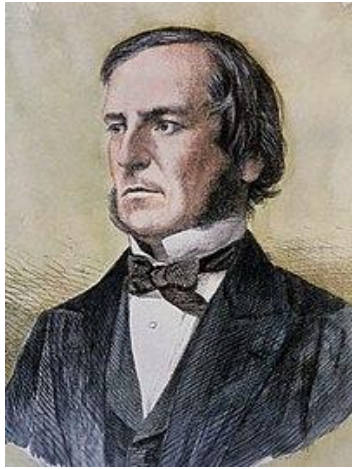
- nodes - wires classified as input, output and internal
- elements - themselves are circuits, with inputs, output and specifications.



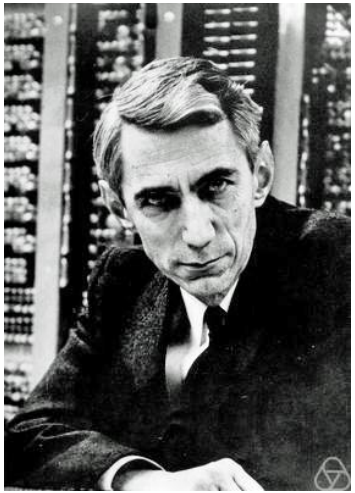
Agenda

- Boolean algebra and logic gates
- Sum of products and product of sums
- Bubble push
- Karnaugh Maps

Boolean algebra and logic gates



George Boole, 1815-1864.
The founder of Boolean Algebra



Claude Elwood Shannon, 1916-2001.
“The father of information theory”

Boolean algebra

Boolean algebra is based on three operations:

- conjunction (and) : $p \wedge q$; p x q; p ▪ q;
- disjunction (or) : $p \vee q$; p + q;
- negation (not) : $\neg p$;

and two constants:

- true : 1
- false : 0

Boolean algebra

Let p, q, and r be three boolean variables.

- Commutative laws:
  - $p \times q = q \times p$
  - $p + q = q + p$
- Associative laws:
  - $(p \times q) \times r = p \times (q \times r)$
  - $(p + q) + r = q + (p + r)$
- Distributive laws:
  - $p \times (q + r) = (p \times q) + (p \times r)$
  - $p + (q \times r) = (p + q) \times (p + r)$

Boolean algebra

Identities: p x 1 = p; p + 0 = p;

annihilation: p x 0 = 0; p + 1 = 1;

Idempotence: p x p = p; p + p = p;

Absorption: p x (p + q) = p; p + (p x q) = p;

Complementation: p x ¬p = 0; p + ¬p = 1;

Double negation: ¬ ¬ p = p;

Boolean algebra

De Morgan duality:

- $\neg p \times \neg q = \neg (p + q)$
- $\neg p + \neg q = \neg (p \times q)$

Sum-of-products form

The **Sum of Products** (SOP) is a common form of expressing Boolean functions in Boolean algebra. It's a way to represent a logical function as a sum (OR operation) of multiple product terms (AND operations).

- It represents the function F by the **cases** of F = 1 (true)
- e.g., $F(A, B, C) = \bar{A}BC + A\bar{B}\bar{C} + ABC$
  - ABC is called minterm, representing a combination of the variables that results in the function F being true.

Sum-of-products form

Every boolean expression can be represented in the SOP form.

- Boolean algebra can always be broken down into fundamental AND, OR, and NOT operations;
- SOP captures this structure by:
  - ANDing the variables or their complements (to form minterms).
  - ORing the minterms together (to account for all rows in which the function is true).
- We can convert a truth table of a function into the form of SOP.

Converting a truth table to the SOP form

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

1. Identify rows where the function is 1
   a. e.g., Row 2 (A=0, B=0, C=1); Row 3, 5, 7;
2. Write minterms for each row
   a. e.g., Row 2: $\bar{A}\bar{B}C$
3. Combine the minterms with OR (+)

$$F(A,B,C) = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$$

Product-of-sums form

The **Product of sums** (POS) is another form of expressing Boolean functions in Boolean algebra. It's a way to represent a logical function as a product (AND operation) of multiple sum terms (OR operations).

- It represents the function F by the cases of F = 0 (false)
- e.g., $F(A, B, C) = (A + B + \bar{C}) \cdot (A + B + C)$
  - *(A+B+C)* is called maxterm, representing a combination of the variables that results in the function F being false.

Converting a truth table to the POS form

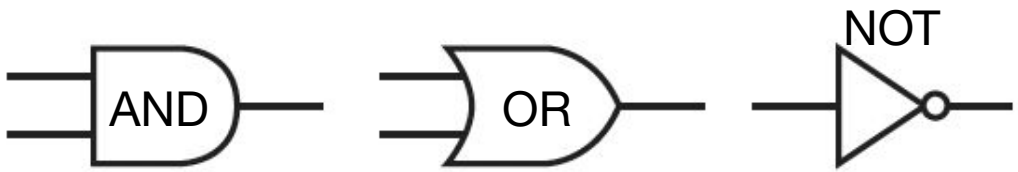
A	B	C	F(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

1. Identify rows where the function is 0
   a. e.g., Row 1 (A=0, B=0, C=0); Row 4, 6, 8;
2. Write maxterms for each row
   a. e.g., Row 1: $(\bar{A} + \bar{B} + \bar{C})$
3. Combine the maxterms with AND

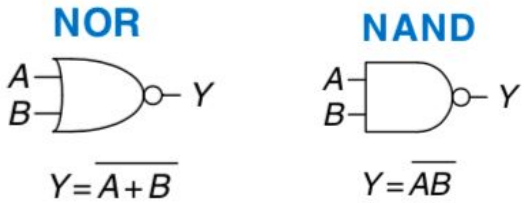
$$F(A,B,C) = (\bar{A} + \bar{B} + \bar{C}) \cdot (\bar{A} + B + C) \cdot (A + \bar{B} + C) \cdot (A + B + C)$$

Logic gates

- A gate is a device that implements basic logic functions, such as AND, OR, NOT (also, called inverse).



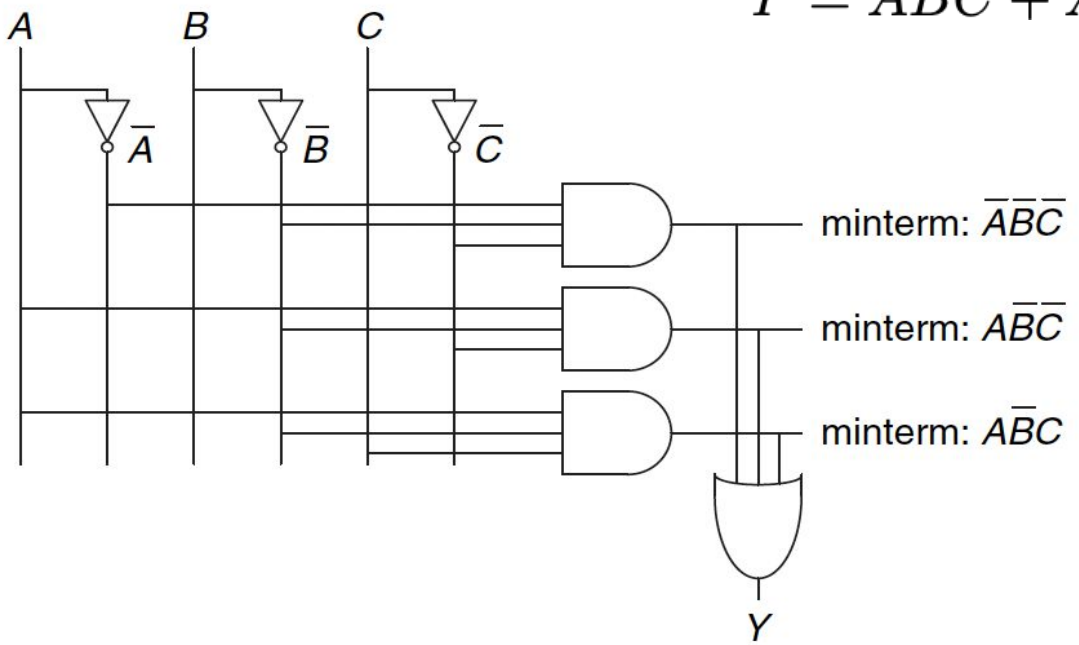
- Any logical function can be constructed using AND gates, OR gates, and NOT.
- In fact, all logic functions can be constructed with only a single gate type, if that gate is inverting.
  - NOR
  - NAND



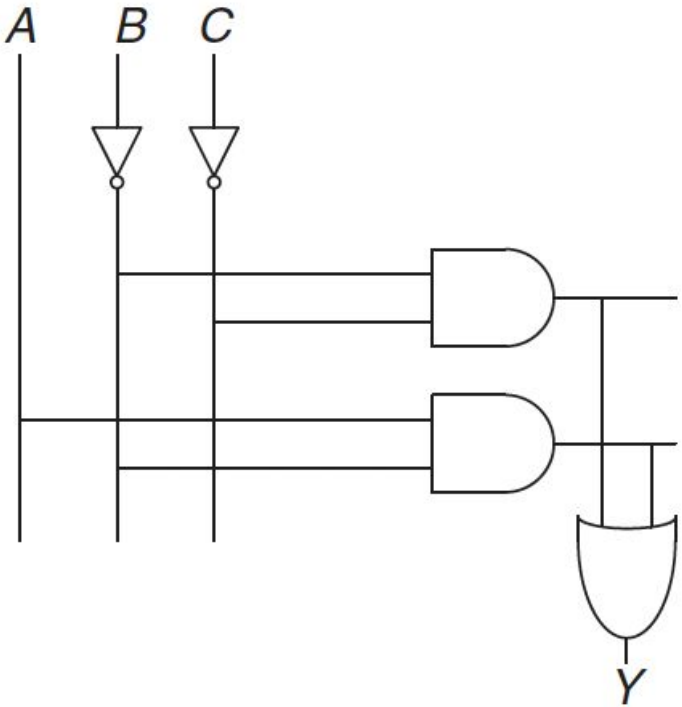
Representing logic functions with gates

sum-of-products in schematic

$$Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$$



Representing logic functions with gates



The previous expression can be simplified to

$$Y = \bar{B}\bar{C} + A\bar{B}$$

Exercise

Show that the equality

$$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C = \bar{B}\bar{C} + A\bar{B}$$

follows from the equations of boolean algebra.

Solution

$$\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} = (A + \overline{A})\,\overline{B}\,\overline{C} \quad \text{distributivity}$$

$$= 1\,\overline{B}\,\overline{C} \quad \text{complementation}$$

$$= \overline{B}\,\overline{C} \quad \text{identity}$$

$$A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C = A\,\overline{B}\,(C + \overline{C}) \quad \text{distributivity}$$

$$= A\,\overline{B}\,1 \quad \text{complementation}$$

$$= A\,\overline{B} \quad \text{identity}$$

Solution

$$\overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C = \overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C$$

by idempotence and associativity

$$= \overline{B}\,\overline{C} + A\,\overline{B}$$

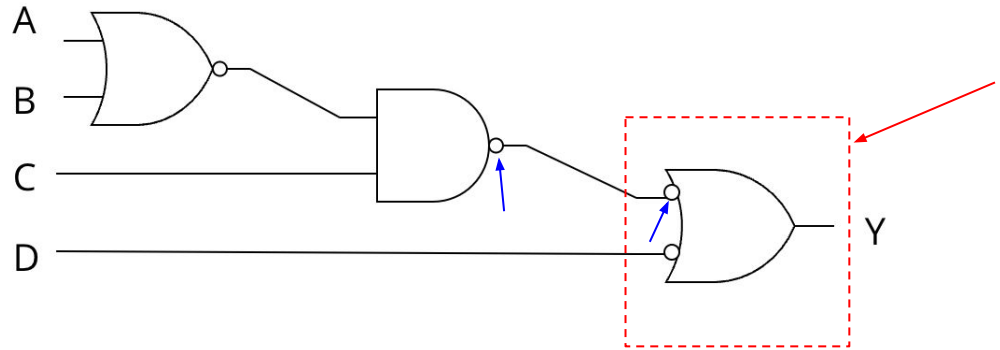
by the two equations of the previous page

Bubble Pushing

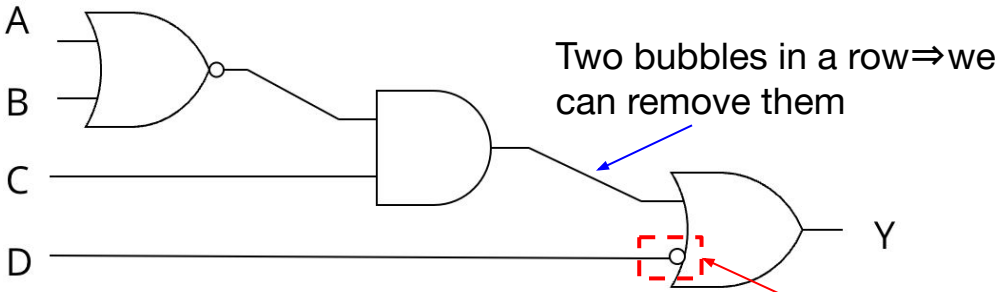
Bubble pushing is a technique to simplify circuit diagrams and optimize logic gate configurations.

- Motivation: circuits prefer NANDs and NORs
- “Bubble” refers to the small circle that denotes an inversion (NOT operation) on the input or output of a logic gate.
- It leverages De Morgan’s Laws,
  - NAND: NOT (A AND B) = (NOT A) OR (NOT B)
  - NOR: NOT (A OR B) = (NOT A) AND (NOT B)

Bubble Pushing



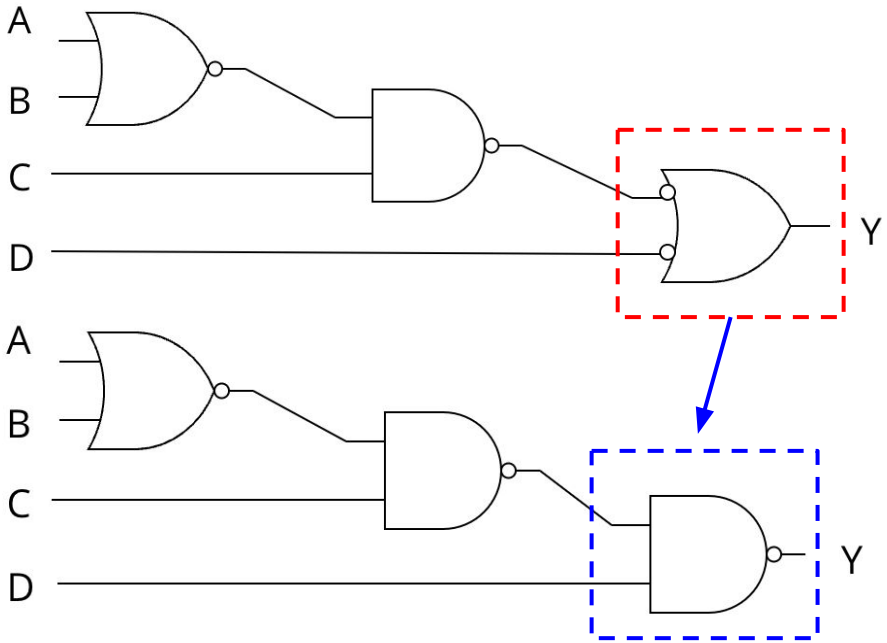
Suppose that one wants to remove the inverse gates at the inputs.



Two bubbles in a row⇒we can remove them

But how to remove this one?

Bubble Pushing

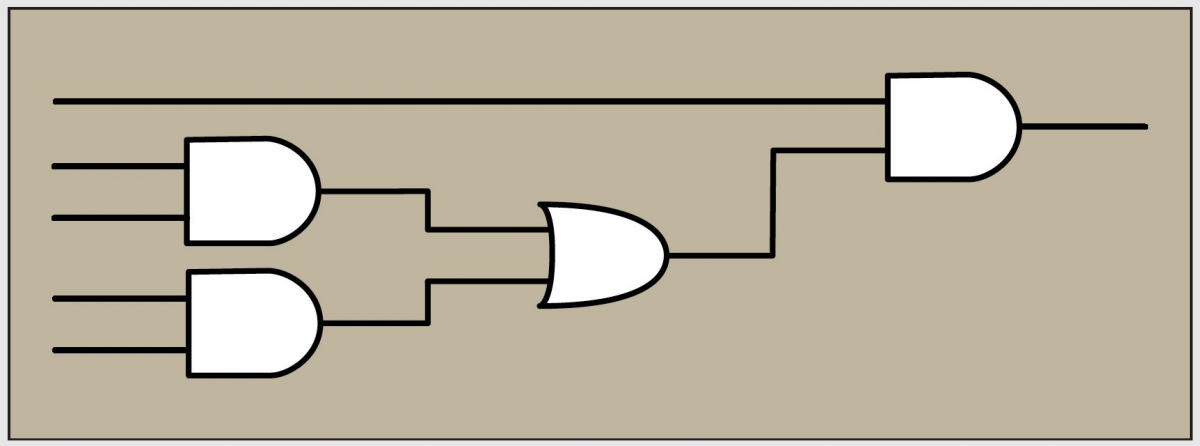


Pushing the bubbles through the OR gate results in an AND gate with the output reversed (i.e., an NAND gate).

- When a bubble is encountered on the input of a logic gate, you can "push" the bubble through the gate, transforming it according to De Morgan's Laws.
- In some situations, one would like to transform a logical circuit expressed with AND and OR gates into an equivalent logical circuit constructed with NAND gates, NOR gates and inverters.
- One typical reason is that NAND, NOR, and inverters are easier to construct in CMOS technology.

Bubble Pushing Example

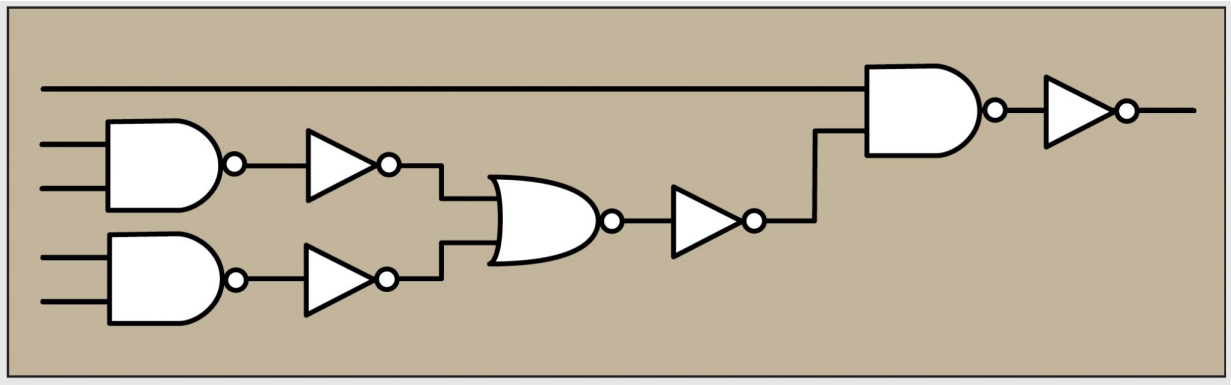
Translate the following circuit into one that only use NAND, NOR, and inverters.



Bubble Pushing Example

One brutal way to achieve the translation is to replace:

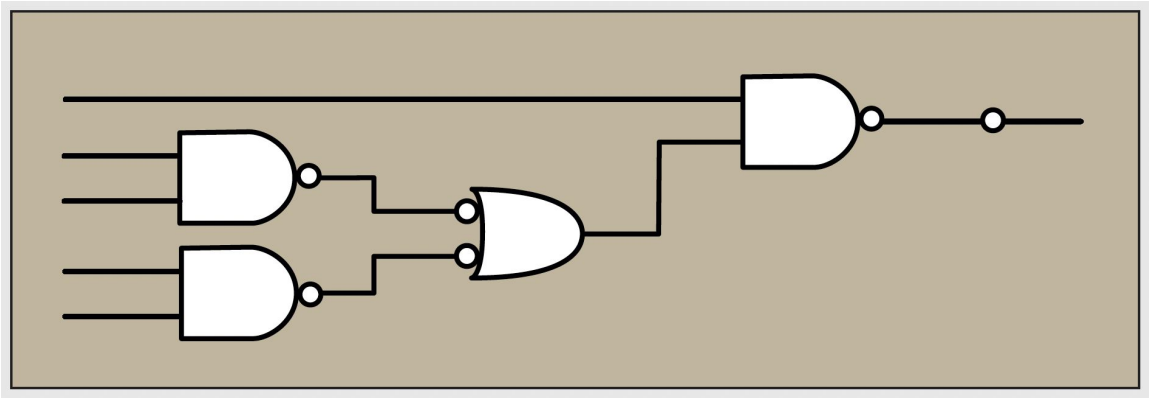
- every AND gate by NAND gate followed by an inverter
- every OR gate by a NOR gate followed by an inverter



This procedure works but is far from optima in general in the number of logical gates in the final circuit.

Bubble pushing Example

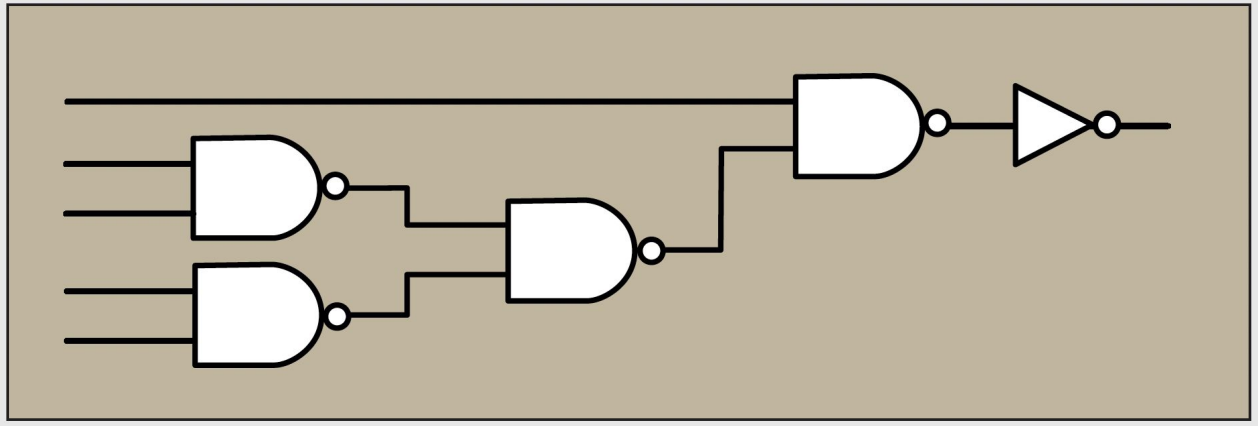
A more sophisticated way to achieve the translation is to add two negations on some of the wires of the original logical circuit:



Note that each negation is represented here by a bubble.

Bubble pushing Example

A more sophisticated way to achieve the translation is to add two negations on some of the wires of the original logical circuit:

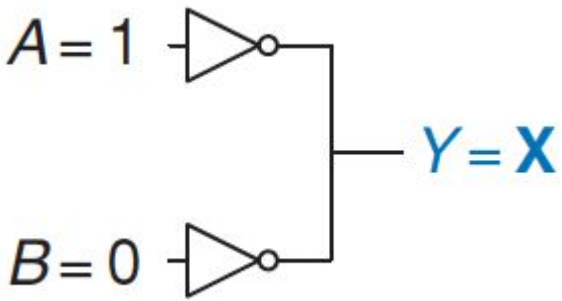


One obtains in this way a CMOS circuit with five logical gates (instead of eight gates as in the previous translation)

In real circuits: illegal and floating values

- Boolean algebra is limited to 0's and 1's.
- However, in real circuits can also have illegal and floating values.
  - We often use X and Z to represent them.

illegal value X

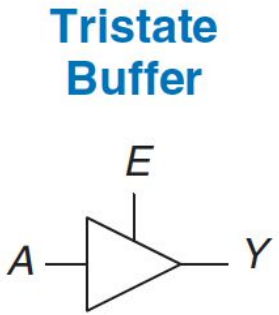


In this circuit, the output Y has an unknown or illegal value.

- When A and B are different (i.e., HIGH and LOW) at the same time, Y is unknown or illegal.
- This situation is, called **connection**, considered as error and should be avoided.
- “contention” may cause large amounts of power to be dissipated between the logical gates, resulting in the circuit getting hot and possibly damaged.

The floating value Z

- Symbol Z indicates a node is driven neither by HIGH nor LOW.
- In practice, a floating value might be 0, might be 1, or might be some voltage in between, depending on the history of the system.
- Tristate buffer is a circuit that can have three possible outputs:
  - HIGH (1)
  - LOW (0)
  - Floating (Z)

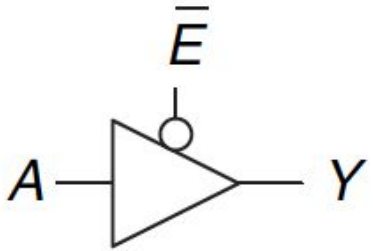


E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Z means Y may receive any voltage depending on the context. This "third state" essentially **turns the output off**, so it does not affect other signals on a shared line.

The floating value Z

- Tristate buffer is a circuit that can have three possible outputs:
  - HIGH (1)
  - LOW (0)
  - Floating (Z)



$\overline{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Tristate buffer that is active low enable. It is like a switch controlled by the E.

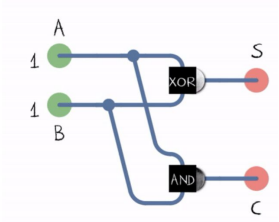
Karnaugh Maps

Karnaugh Map (K-Map): is a graphical tool used to simplify Boolean equations.

- It was introduced in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs.
- Work well with small number of variables (<5)
- Given a truth table, we can use K-Map to find the corresponding Boolean equations.

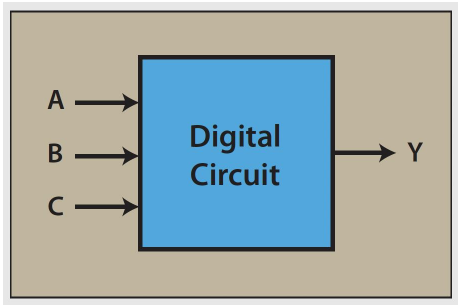
Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S = A **XOR** B
C = A **AND** B



A typical truth table

- There is a circuit and its truth table, and we want to find the equivalent Boolean expression.

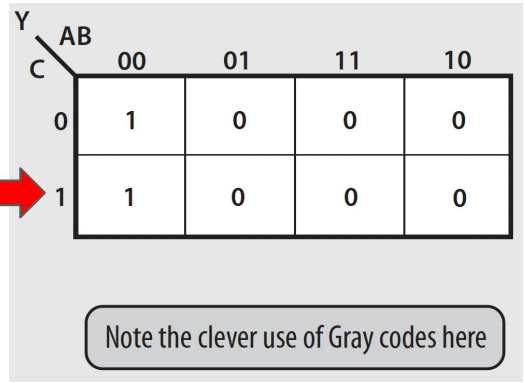


A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

From the truth table to the K-map

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

A truth table lists all possible input combinations for a Boolean expression and their corresponding output values.

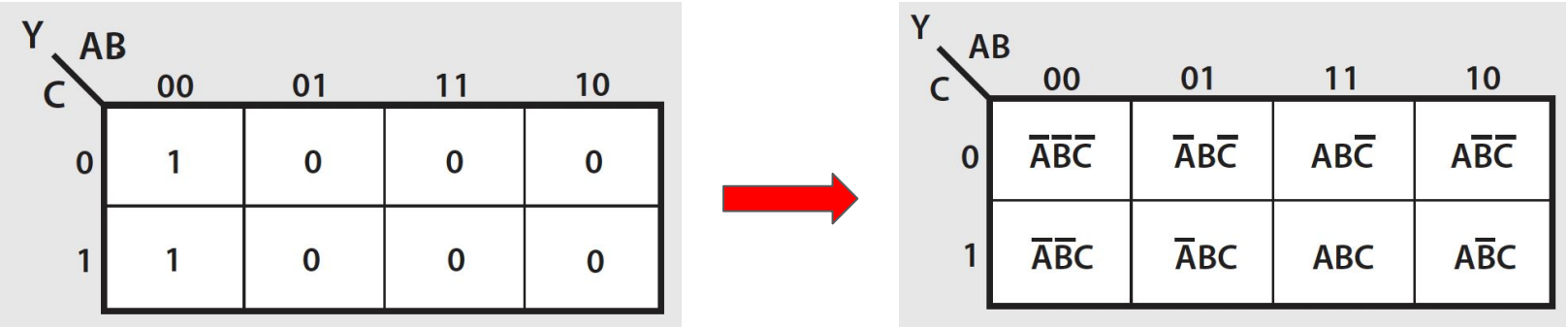


A K-map organizes these input combinations into a grid format where each cell represents a unique combination of input variables.

K-map:

- Rows and columns represent a variable (or a combination of some variables) and should be arranged as the Gray codes.
- Gray codes: a binary numerical system where two successive values differ in only one bit. (e.g., the columns are 00 01 11 10 in the left K-map.)
- The K-map has 2^n cells where n is the number of variables. (each cell represents a row in truth table)
- Each 1 in the K-map indicates the Boolean expression outputs true for that particular combination.

Same K-map with associated minterms

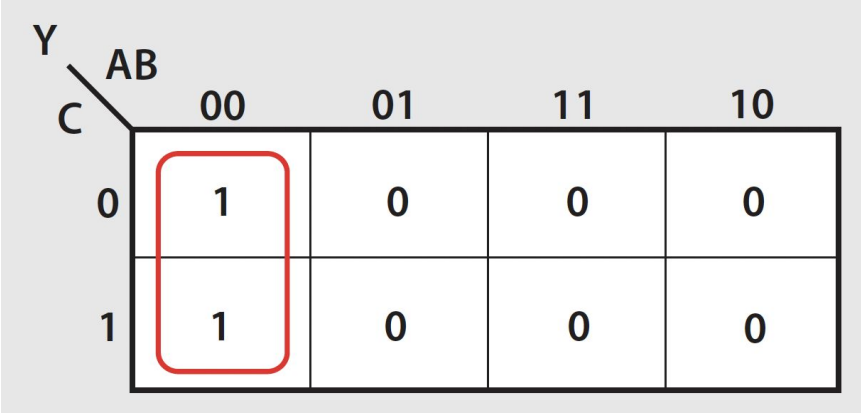


Y / C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

From this, one deduces that:

$$Y = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C$$

K-map minimisation



- K-map provide an easy visual way to minimize logic.
- By circling the 1s in adjacent squares, one deduces that:

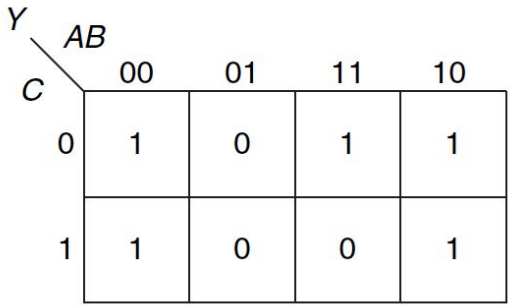
$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C = \overline{A}\,\overline{B}$$

Logic minimization with K-map

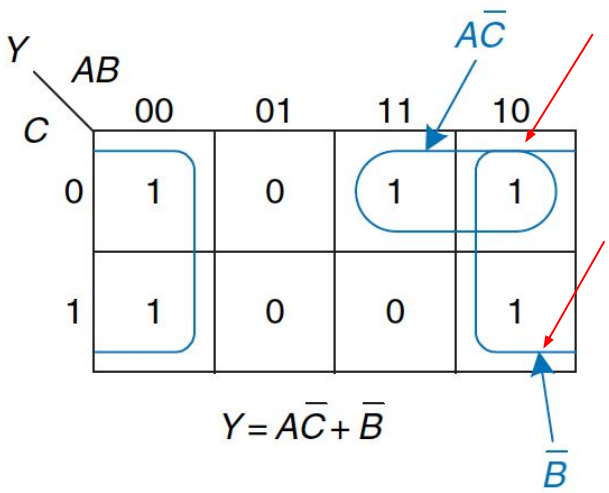
Rules for finding a minimized equation from a K-map:

- Use the fewest circles to cover all the 1’s.
- All the squares in each circle must contain 1’s.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible. (i.e., a circle should be a prime implicant.)
    - A prime implicant is a group of '1's on the K-map that cannot be expanded further without losing the ability to cover those minterms.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

Logic minimization with K-map Example



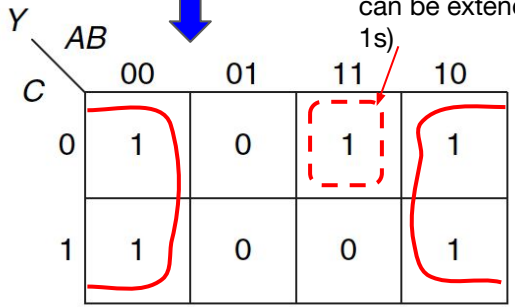
correct



The circle should be as large as possible. A 1 can be circled multiple times

Can wrap around the edges

Wrong



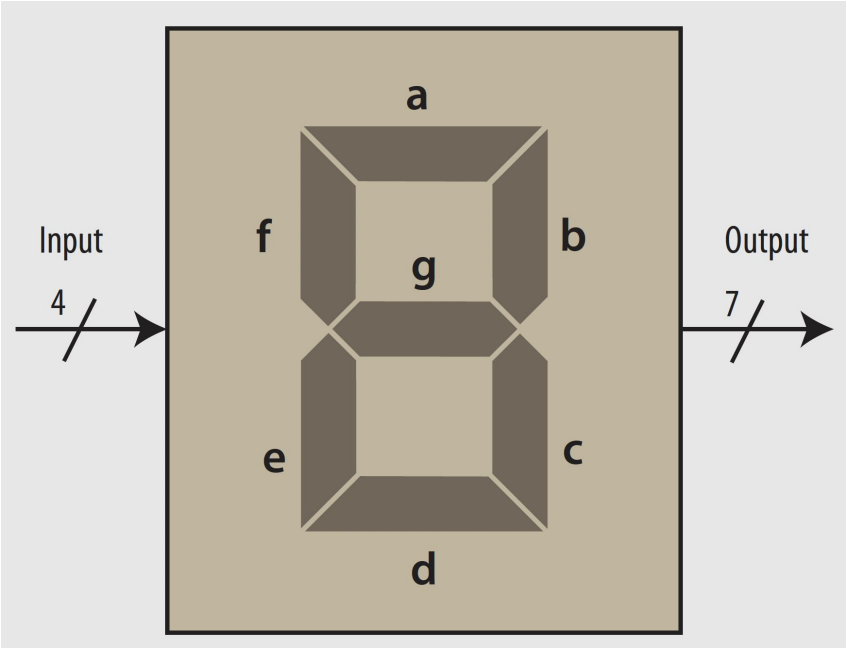
not a prime implicant (because it can be extended to include more 1s)

- Each circle is a sum of products. Since the cells are arranged in Gray code, there are complements between every two cells.(e.g., column 11 and 10)

Seven-segment display digits

- Seven-segment displays (7sd) are typically used in watches, calculators, and instruments to display decimal data.
- Seven-segment displays (7sd) are some of the most common electronic display devices in use.

Seven-segment display decoder



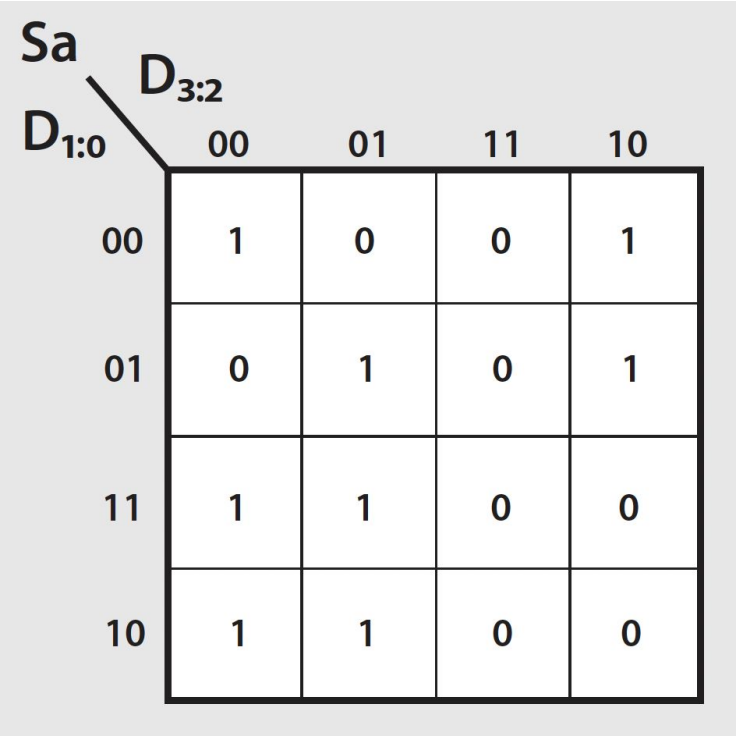
- By lighting up a specified pattern of 7 LEDs, this type of decoder can create numbers 0-9 for digital display.
- The circuit has an input of 4 variables and 7 outputs.
- S_a, S_b , …, S_g represent the 7 outputs

Seven-segment display decoder truth table

$D_{3:0}$	Sa	Sb	Sc	Sd	Se	Sf	Sg
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
others	0	0	0	0	0	0	0

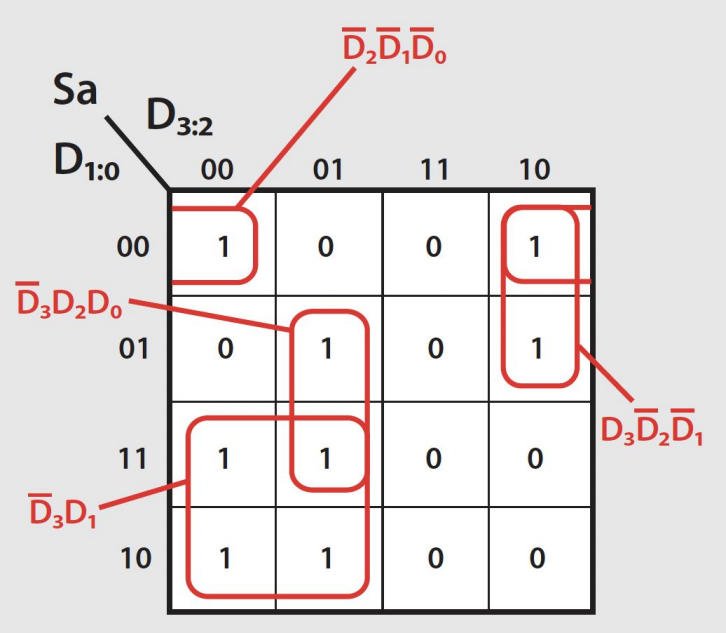
Each output (i.e., S_a, S_b , …, S_g) is an independent function of input four variables.

Decoder K-maps minimisation



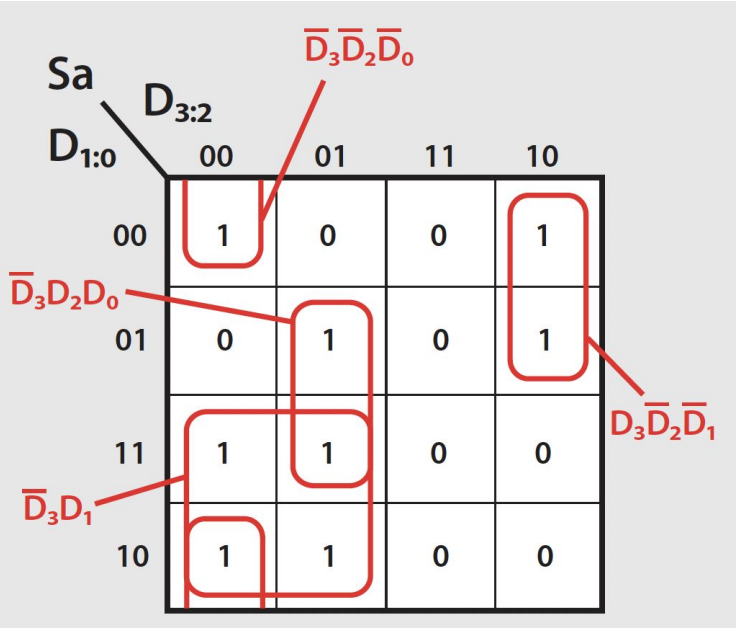
Sa / $D_{3:2}$ \ $D_{1:0}$	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0

Decoder K-maps minimisation



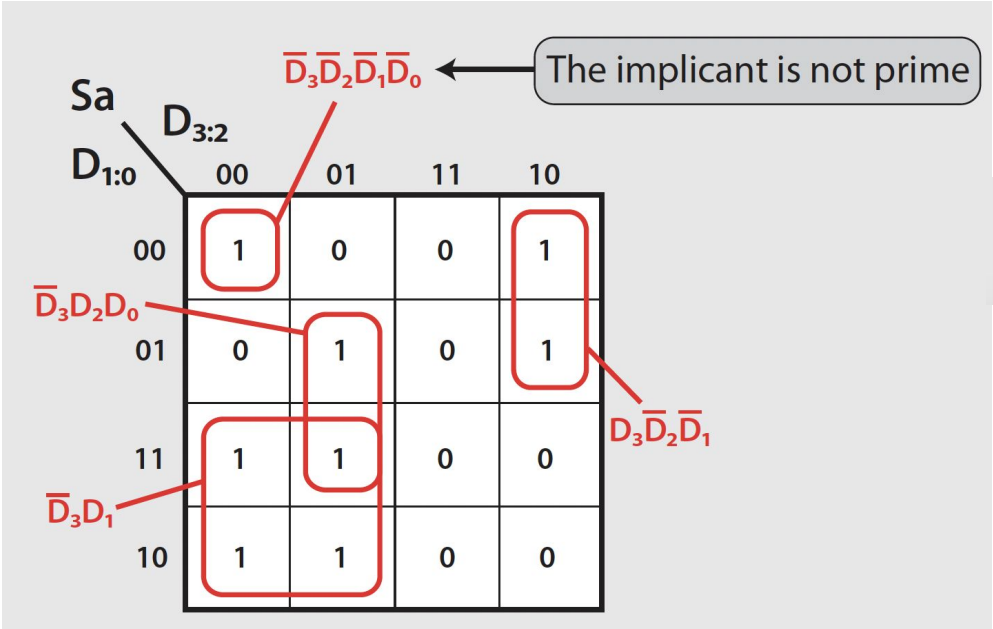
$$Sa = \overline{D}_3 D_1 + \overline{D}_3 D_2 D_0 + D_3 \overline{D}_2 \overline{D}_1 + \overline{D}_2 \overline{D}_1 \overline{D}_0$$

Decoder K-maps minimisation



$$Sa = \overline{D}_3 D_1 + \overline{D}_3 D_2 D_0 + D_3 \overline{D}_2 \overline{D}_1 + \overline{D}_3 \overline{D}_2 \overline{D}_0$$

A common mistake

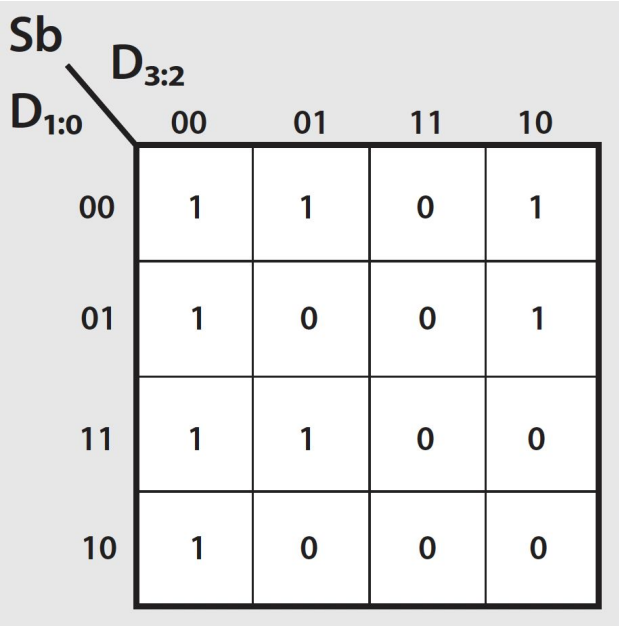


$$Sa = \overline{D}_3D_1 + \overline{D}_3D_2D_0 + D_3\overline{D}_2\overline{D}_1 + \overline{D}_3\overline{D}_2\overline{D}_1\overline{D}_0$$

Given D_1, the values of two minterms are opposite.

If you include a non-prime implicant in your K-map, the resulting Boolean expression will likely be more complex than necessary, leading to inefficiencies in the digital circuit.

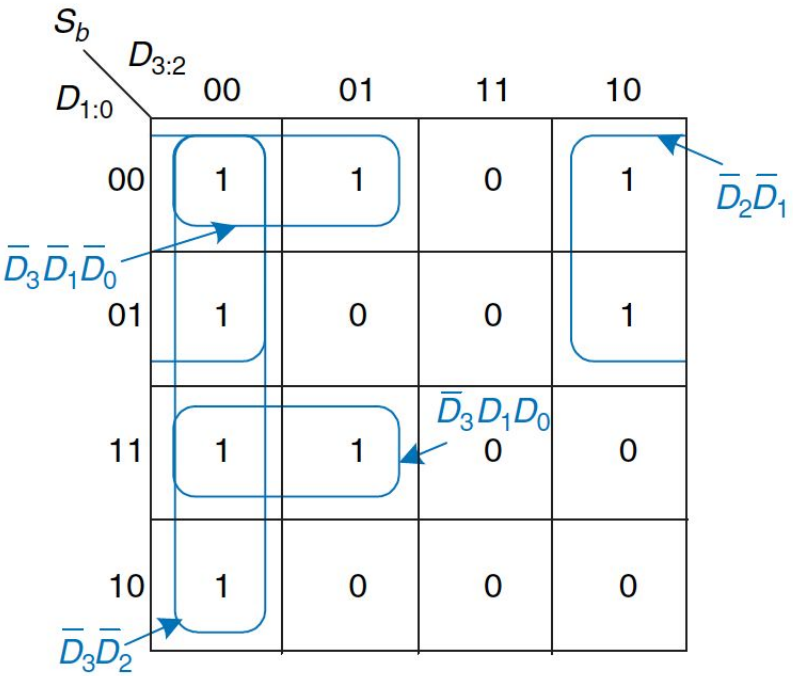
Exercise 1



Sb: $D_{1:0}$ \ $D_{3:2}$	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

Using K-map to minimize the S_b.

Answer: (exercise 1)



$$S_b = \bar{D}_3\bar{D}_2 + \bar{D}_2\bar{D}_1 + \bar{D}_3D_1D_0 + \bar{D}_3\bar{D}_1\bar{D}_0$$

Truth table with Don't Cares

$D_{3:0}$	Sa	Sb	Sc	Sd	Se	Sf	Sg
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
others	X	X	X	X	X	X	X

Here, the value **X** means that the value is undetermined : it can be 0 or 1.

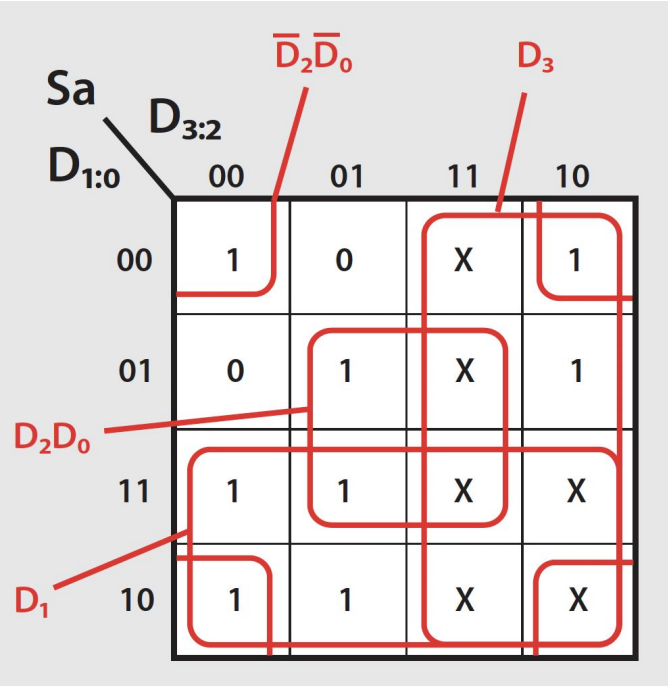
- "***Don't Cares***": the output of don't care what the inputs are (here, it is the row "others").
- We use symbol **X** to describe inputs that the output doesn't care about.
- **X** can be 0 or 1.

K-maps minimisation with X's

Sa / $D_{3:2}$ / $D_{1:0}$	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X

- X's can be circled if they help cover the 1's with fewer or larger circles.
- but they do not have to be circled if they are not helpful.

K-maps minimisation with X's



$$Sa = D_3 + D_2D_0 + \overline{D}_2\overline{D}_0 + D_1$$

- X's can be circled if they help cover the 1's with fewer or larger circles.
- but they do not have to be circled if they are not helpful.

Exercise 2

Using K-map to design the boolean equations of the half-adder.

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Exercise 3

Complete the design of the seven-segment decoder by designing boolean equations for the segments S_c and S_d:

- assuming that inputs greater than 9 must product blank (0) outputs
- assuming that inputs greater than 9 are don’t cares.

Then, sketch a reasonably simple gate-level implementation in the case b and simulate the resulting circuits on CircuitLab(https://www.circuitlab.com/).