Week 0100
Logic circuit design and Karnaugh Maps

CCS1310

Computer Systems Architecture

Dr. Kostas Dimopoulos

OVERVIEW OF THE LECTURE

- Universal gates
- Combinatorial logic design with Boolean functions
- Simplifying logic design with K-maps

Universal Gates

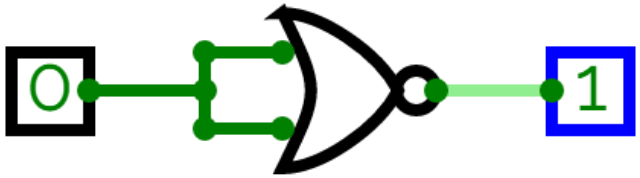
The NOR as a universal gate

- It is possible to make NOT, OR and AND with only using NOR gates
- This is extremely useful as it is vastly easier to manufacture chips that contain multiple gates of the same type, rather than chips with different types of gates
- NOR is OR with a NOT :
  - A NOR B = (A+B)’

NOT from NOR

- A NOT gate can be implemented by passing the same input into both inputs of the NOR gate.

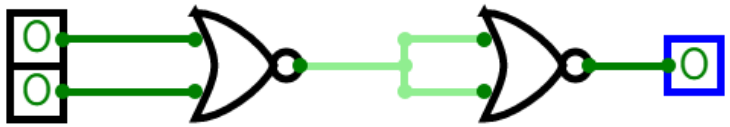
A	B	NOR(A,B)
0	0	1
0	1	0
1	0	0
1	1	0



OR from NOR

- As the NOR is the opposite of OR, an OR gate can be implemented by passing the output of NOR to the NOT gate implemented earlier.

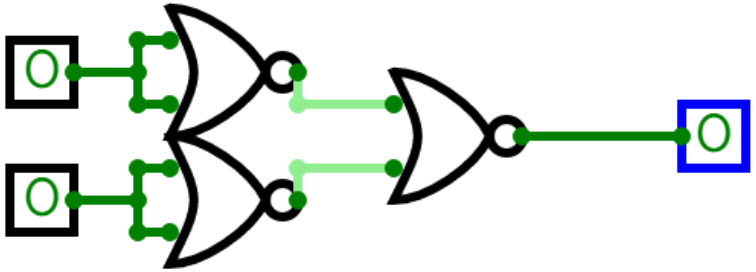
A	B	NOR(A,B)	OR(A,B)
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1



AND from NOR

- Since the NOR gate outputs true only when both inputs are 0, an AND gate can be implemented by inverting the inputs to a NOR gate.

A	B	NOR(A,B)	AND(A,B)
0	0	1	0
0	1	0	0
1	0	0	0
1	1	0	1



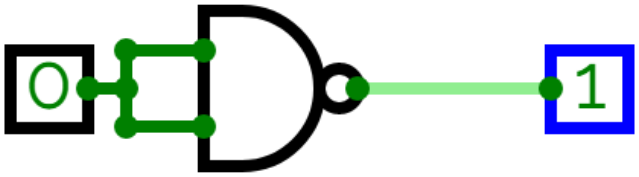
NAND as a universal gate

- It is possible to make NOT, OR and AND with only using NAND gates
- This is extremely useful as it is vastly easier to manufacture chips that contain multiple gates of the same type, rather than chips with different types of gates
- NAND is AND with a NOT after:
  - A NAND B = (A.B)’

NOT from NAND

- Similarly to NOR, a NOT gate can also be implemented by joining the inputs of a NAND gate.

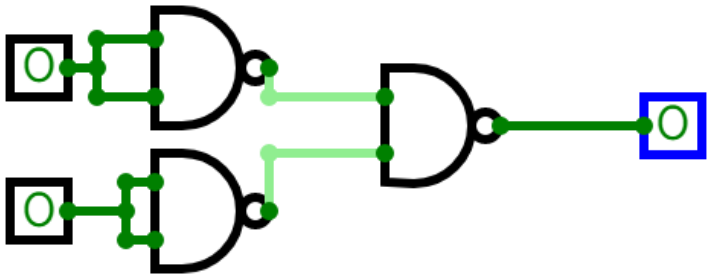
A	B	NAND(A,B)
0	0	1
0	1	1
1	0	1
1	1	0



OR from NAND

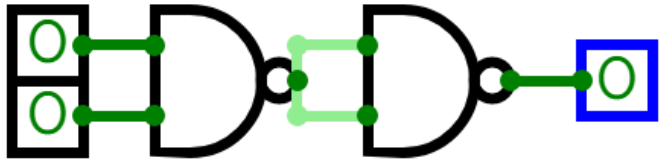
- The only time the NAND gate output is 0 is when both inputs are 1. Therefore, by inverting the inputs of a NAND gate, an OR gate can be implemented.

A	B	NAND(A,B)	OR(A,B)
0	0	1	0
0	1	1	1
1	0	1	1
1	1	0	1



AND from NAND

- The AND gate is simply the output of a NAND gate inverted.



Building a digital circuit with only NANDs

Building a digital circuit with only NANDs

- For any give Boolean expression F(…):

1. Double negate it
2. Apply De Morgan's Law to the inner bracket
3. Construct the NAND circuit remembering that:
   - A NAND B = (A.B)'

- Example let F(A, B, C, D) = AB' + C'D

1. Double negation

- Since the NAND gate is a combination of a NOT gate and an AND gate, we first apply a double negation to the entire expression so that we are able to standardize it later on.
- Adding a double negation does not alter the inherent value of the expression as a double negation always nullifies itself.
- F = (F')' = ((AB' + C'D)')'

2. Apply De Morgan's Law

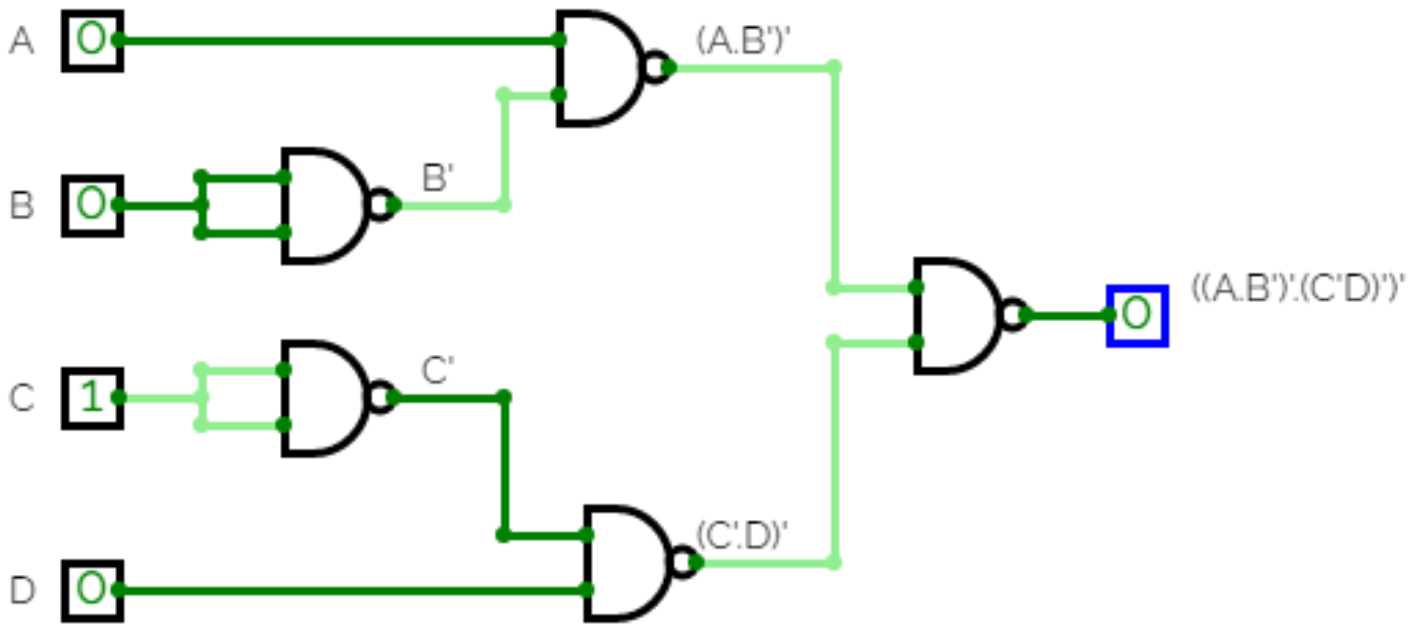
- We first apply De Morgan's Law to the innermost bracket, such that we preserve the outermost negation at the time of expressing the F as a NAND expression.
- Thus, by applying De Morgan's Law:
  - F = ((AB' + C'D)')' = ((AB')' . (C'D)')'
- The Boolean expression is now standardized such that it can completely be represented by a NAND gate at every input level.

3. Construct the NAND circuit

- Now that you have gotten the Boolean expression to the required standard, you can implement it as a NAND circuit.
- Remember that x NAND y = (x.y)’
- F = ((AB')' . (C'D)‘)‘
- F = (A NAND B') NAND (C' NAND D)
- F = (A NAND (B NAND B)) NAND ((C NAND C) NAND D)

3. Construct the NAND circuit

- F = (A NAND (B NAND B)) NAND ((C NAND C) NAND D)



Building a digital circuit with only NORs

Building a digital circuit with only NORs

- For any give Boolean expression F(…):

1. Double negate it
2. Apply De Morgan's Law to the inner bracket
3. Construct the NOR circuit remembering that:
   - A NOR B = (A+B)'

- Example let F(A, B, C, D) = (A + B').(C' + D)

1. Double negation

- Since the NOR gate is a combination of a NOT gate and an OR gate, we first apply a double negation to the entire expression so that we are able to standardize it later on.
- Adding a double negation does not alter the inherent value of the expression as a double negation always nullifies itself.
- F = (F')' = (((A + B').(C' + D))')'

2. Apply De Morgan's Law

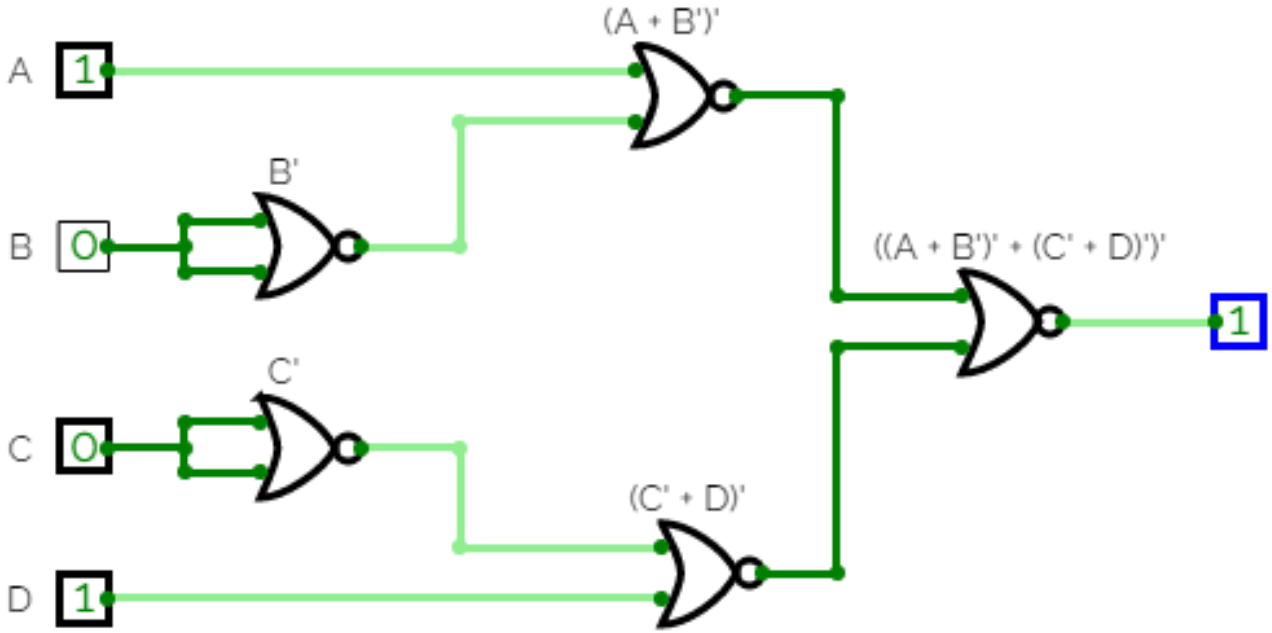
- We first apply De Morgan's Law to the innermost bracket, such that we preserve the outermost negation at the time of expressing the F as a NOR expression.
- Thus, by applying De Morgan's Law:
  - F = (((A + B').(C' + D))')' = ((A + B')' + (C' + D)')'
- The Boolean expression is now standardized such that it can completely be represented by a NOR gate at every input level.

3. Construct the NAND circuit

- Now that you have gotten the Boolean expression to the required standard, you can implement it as a NOR circuit.
- Remember that x NOR y = (x+y)’
- F = ((A + B')' + (C' + D)')'
- F = (A NOR B') NOR (C' NOR D)
- F = (A NOR (B NOR B)) NOR ((C NOR C) NOR D)

3. Construct the NAND circuit

- F = (A NAND (B NAND B)) NAND ((C NAND C) NAND D)

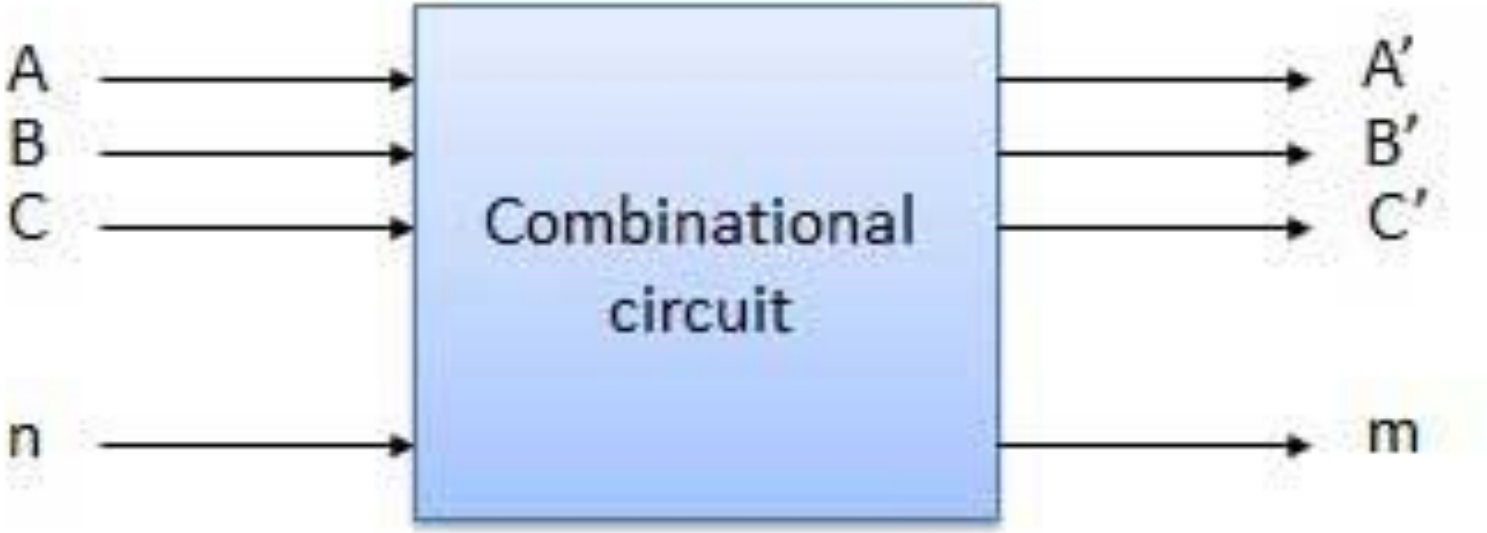


Logic design

Combinational Logic

What is logic design

- The process of obtaining a digital circuit from a functional definition. It also involves some optimisation methods to reduce the design before implementing it as a circuit.



Design procedure of combinational circuits

1. Find the required number of input variables and outputs from given specifications.
2. Formulate the Truth table. If there are ‘n’ input variables, then there will be 2^n possible combinations. For each combination of input, find the output values.
3. Find the Boolean expressions for each output. If necessary, simplify those expressions.
4. Implement the above Boolean expressions corresponding to each output by using Logic gates.

Boolean Functions Implementation

Any Boolean function can be implemented in electronic form as a network of gates. Consider the following truth table:

A	B	C	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**	**1**
1	1	1	0

We can express F by simply itemizing the combination of values A, B, C that cause F to be 1:

$$F = \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C}$$

This is called the **Sum-of-Products** (SOP) form. The SOP form expresses that the output is 1 if any of the input combinations that produce 1 is true.

Boolean Functions Simplification

- The SOP derives very long expressions for complex functions
- Various techniques can be used to minimise/simplify these expressions
- One such method widely used are the **Karnaugh maps**

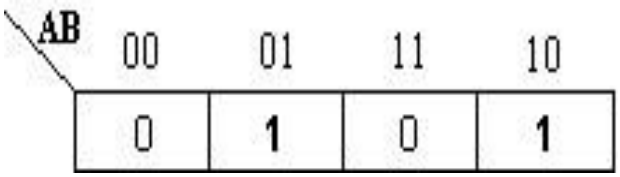
Karnaugh Maps

- Karnaugh maps are a convenient way of representing a Boolean function of a small number (4-6) of variables
- Map: An array of 2^n squares, representing the possible combinations of values of n binary variables

TRUTH TABLE

A	B	F
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

KARNAUGH MAP



BOOLEAN FUNCTION

$$F = A\overline{B} + \overline{A}B$$

Always list combinations in order: 00, 01, 11, 10

We have two variables (A, B) so we have four cells in the K-map

Constructing a K-map: Example 2

TRUTH TABLE

A	C	D	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**	**1**
1	1	1	0

KARNAUGH MAP

A \ CD	00	01	11	10
0	0	0	**1**	**1**
1	0	0	0	**1**

BOOLEAN FUNCTION

$$F = \overline{A}C\overline{D} + \overline{A}CD + AC\overline{D}$$

Always list combinations in order: **00, 01, 11, 10**

We have three variables (A, B, C) so we have eight cells in the K-map

Constructing a K-map: Example 3

KARNAUGH MAP

AB \ CD	00	01	11	10
00	0	0	**1**	0
01	0	0	0	0
11	**1**	0	0	0
10	0	**1**	0	0

BOOLEAN FUNCTION

$$F = \overline{A}\,\overline{B}CD + A\overline{B}\,\overline{C}D + AB\overline{C}\,\overline{D}$$

We have four variables (A, B, C, D) so we have sixteen cells in the K-map

Always list the combinations in order: **00, 01, 11, 10**

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

Minimization using a K-map

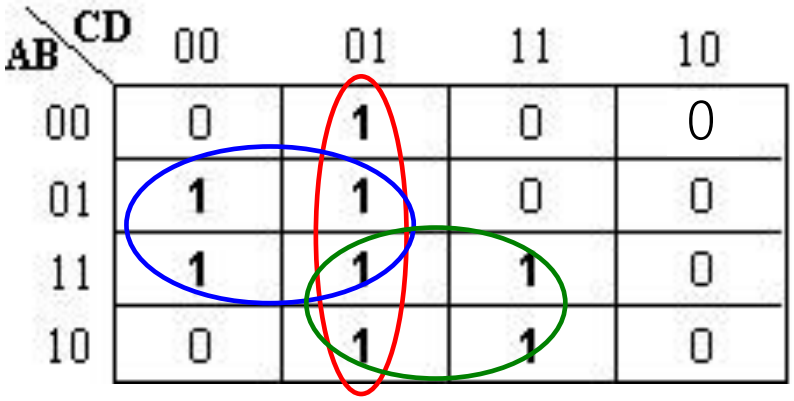
- In any two adjacent squares being 1, the corresponding product terms **differ in only one variable**
  - **Merge the terms by eliminating this 'varying' variable**
- Adjacency includes wrapping around the edge of the map; top adjacent to bottom, left adjacent to right
- Group any $\mathbf{2^n}$ adjacent squares
- Always look for the largest possible grouping
- For 1's that remain ungrouped repeat the process looking for successively smaller groupings
- If any isolated 1's remain after the groupings then each of these is circled individually
- Any group of 1's that is completely overlapped by other groups can be eliminated
  - it is redundant and may be ignored

Look at examples in: Kmap_examples.pdf on our website

Minimization: Example 1

- Consider the truth table on the right
- The minimized expression using a Karnaugh map is: $B\overline{C} + \overline{C}D + AD$

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

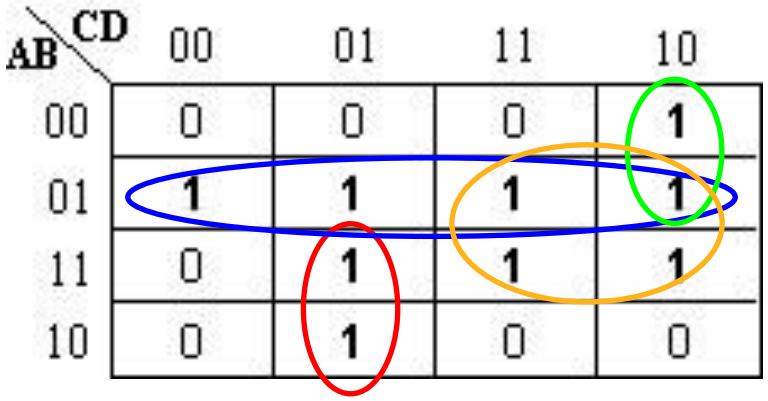


- when the sum of products is:

$$\overline{A}\overline{B}\overline{C}D + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + A\overline{B}\overline{C}D + A\overline{B}CD + AB\overline{C}\overline{D} + AB\overline{C}D + ABCD$$

Minimization: Example 2

- Consider the truth table on the right
- As you understand, the sum of products will be very long since the function becomes TRUE (1) in 9 cases
- We construct the K-map:



- The minimized expression is:

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

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