SCC131: Digital Systems

Topic 4: Boolean logic

Boolean (a.k.a. binary) logic

- The fundamental binary logical operators are **AND**, **OR**, **NOT** and **XOR**

- They are very commonly seen in programming languages:

  - e.g., conditional and looping statements in Java or C
    - `if (expression && ...) ...  /* && is logical AND */`
    - `while (expression || ...) ... /* || is logical OR */`
    - etc.

  - e.g., bit manipulation in C
    - `byte1 = byte1 & 0x7f;` // Clear top bit of byte (n.b. **& is *bitwise* AND)**
    - `byte2 = byte2 | 0x80;` // Set top bit of byte (n.b. **| is *bitwise* OR)**

- But boolean logic is also crucial to the design of computer hardware at the lowest level…

Boolean logic operations can be understood as "truth tables"

- For any pair of binary digits (bits) A and B…

AND

A	B	Q
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

TRUE if, and only if, all inputs are TRUE

OR

A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

TRUE if any input is TRUE

NOT

A	Q
0	**1**
1	**0**

TRUE if, and only if, the single input is FALSE

XOR

A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

(eXclusive OR)

True if an odd number of inputs are TRUE, otherwise FALSE

1 is interpreted as meaning TRUE; and 0 as FALSE

Notation

- Instead of AND/OR/NOT/XOR you will often see
  - AND: • or $\wedge$ or *<nothing>*
    cf. "product"
  - OR: **+** or $\vee$
    cf. "sum"
  - NOT: **’** or **¯** *(bar)* or $\neg$
    ’ may be pronounced as *prime*
    ¯ may be pronouced as *bar*
  - XOR: $\oplus$

Note also the strong relationship to set theory and set operations

...see discrete maths course

Logic components (a.k.a. "logic gates")

AND

A	B	Q
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

OR

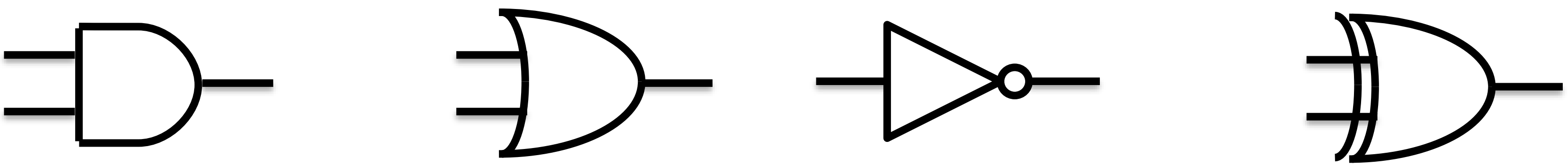
A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

NOT

A	Q
0	**1**
1	**0**

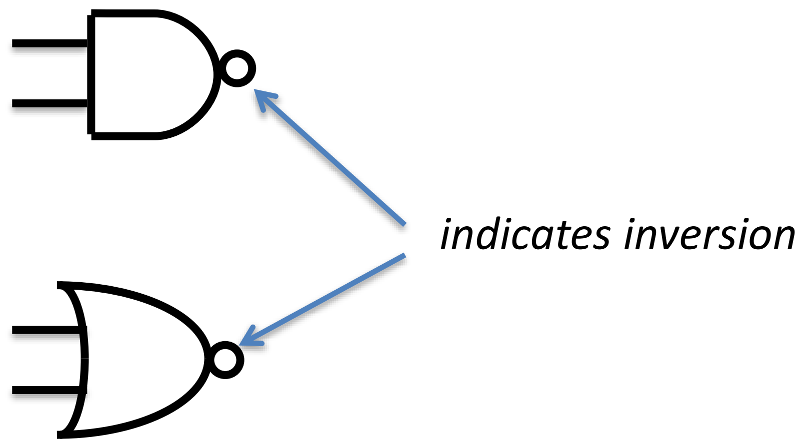
XOR

A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**



Inverted logical operations

- In practice, different gates, called NANDs and NORs, are very commonly used instead of AND/OR/NOT/XOR gates
  - A NAND B = (A AND B)’
  - A NOR B = (A OR B)’
  - NOT AND → **NAND**
  - NOT OR → **NOR**



- This is because NAND and NOR are “universal”: **any** binary logic circuit can be built entirely from NAND gates, or from NOR gates
  - Also: using only NANDs (or NORs) makes circuit design significantly more cost-effective, as only one type of component is needed

NAND and NOR as truth tables…

AND

A	B	Q
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

OR

A	B	Q
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

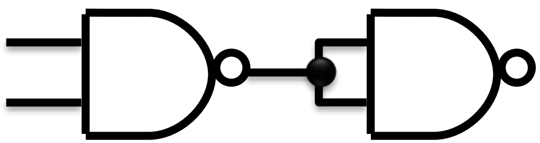
NAND

A	B	Q
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

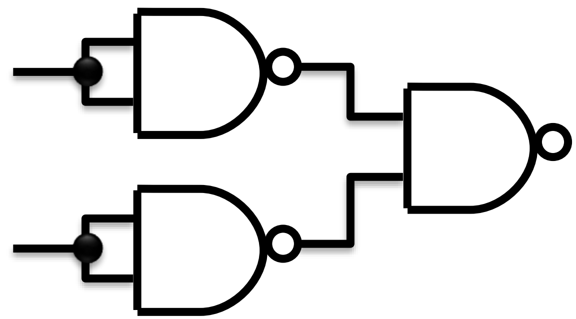
NOR

A	B	Q
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

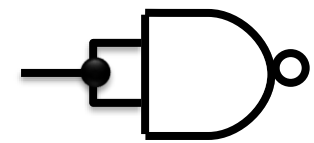
Building AND/OR/NOT from NANDs/NORs



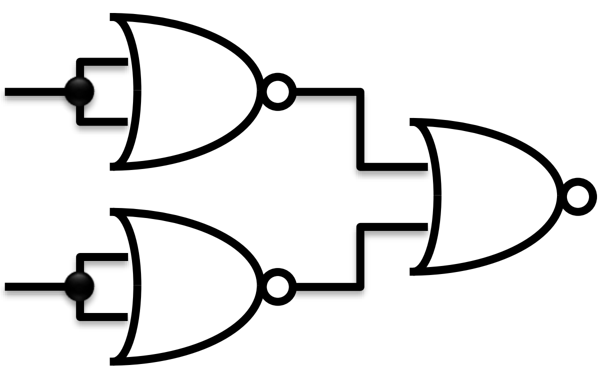
AND



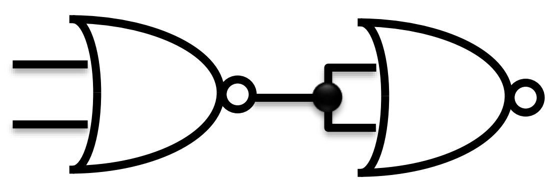
OR



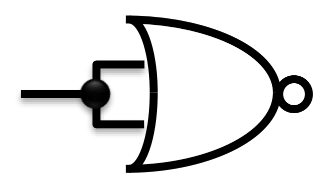
NOT



AND

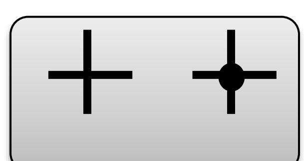


OR



NOT

Crossing wires (no connection)



Junction (connection)

What can we use to build computer logic?

Transistors

Vacuum Tubes

Electro-mechanical Relays

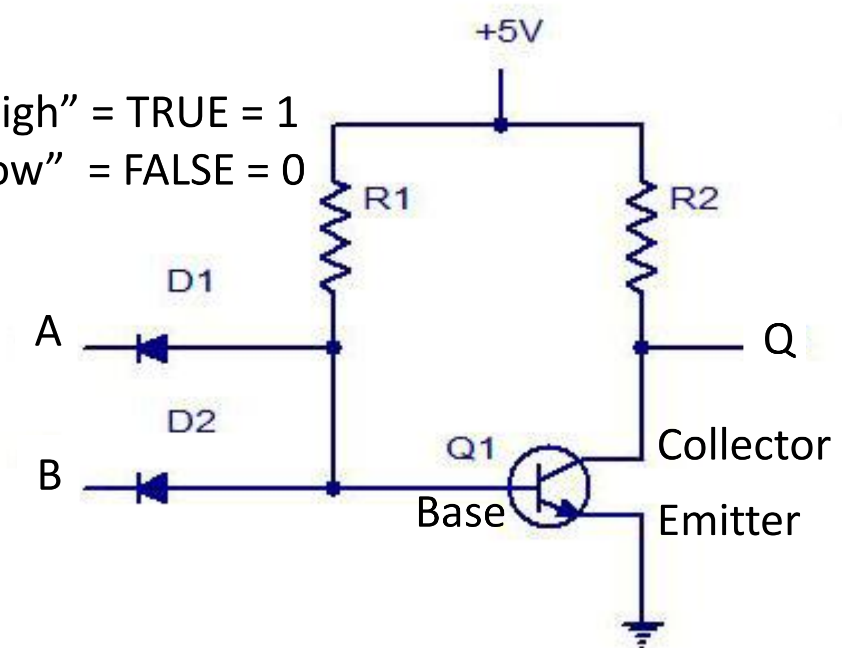
Sheet Metal

...anything we can build a switch from

Example: building a NAND gate from a transistor

- Applying +5v to inputs A and B "opens" the transistor that so current can flow from the collector to the emitter, taking Q down to 0
  - (n.b. this is our sole foray into analog electronics!)

+5v = "high" = TRUE = 1
0v = "low" = FALSE = 0



NAND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

Background

Boolean algebra

Law	AND form	OR form
Identity 1	$A = A''$	$A = A''$
Identity 2	$1A = A$	$0 + A = A$
Null	$0A = 0$	$1 + A = 1$
Idempotence	$AA = A$	$A + A = A$
Complementarity	$AA' = 0$	$A + A' = 1$
Commutativity	$AB = BA$	$A + B = B + A$
Associativity	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributivity	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption	$A(A + B) = A$	$A + AB = A$
*****de Morgan's law*****	$(AB)' = A' + B'$	$(A + B)' = A'B'$

proven on next slide

familiar from School?

Note relationship between AND and OR variants... Swap 0s and 1s, ANDs and ORs ...one function is the *dual* of the other – if a function is correct, its dual must also be

Demonstration of (some of) the laws of Boolean Algebra by *perfect induction*

- That is, by tabulating all possible combinations!

Identity 1

A = A’’		
A	A’	A’’
0	1	0
1	0	1

(OR form of Identity 1 is same as above)

Identity 2

1A = A		
1	A	1 AND A
1	0	0
1	1	1

0 + A = A		
0	A	0 OR A
0	0	0
0	1	1

Null

0A = 0		
0	A	0 AND A
0	0	0
0	1	0

1 + A = 1		
1	A	1 OR A
1	0	1
1	1	1

Idempotence

AA = A		
A	A	A AND A
0	0	0
1	1	1

A + A = A		
A	A	A OR A
0	0	0
1	1	1

Complemen -tarity

AA’ = 0		
A	A’	A AND A’
0	1	0
1	0	0

A + A’ = 1		
A	A’	A OR A’
0	1	1
1	0	1

Proof of absorption by perfect induction

A (A + B) = A			
A	B	A + B	A (A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

A + AB = A			
A	B	AB	A + AB
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

de Morgan's Law

- $(AB)' = A' + B'$ $(A + B)' = A'B'$ *Very useful*!

- Whenever we see an expression whose sub-expressions are all ANDed together, or all ORed together, we can re-state by
  1. negating the overall expression
  2. negating the sub-expressions
  3. flipping the operators from OR to AND, or vice versa

- Can you demonstrate, using perfect induction (i.e. a truth table), that $(A' + B')' = AB$???

Towards the design of logic circuits

- Here are some examples of logic circuits that we might want to design
  - Traffic lights, counters, clocks, multiplexers, segmented numeric displays, …
  - *Computer ALUs*, *computer control units, computer memory units, buses, etc*…

- The basic approach is
  1. Write out a truth table for the desired logical function
  2. Derive a boolean expression by ORing together all the rows whose "output column" is 1
     - This is often called the ***sum-of-products*** form (cf. arithmetic "+")
  3. Translate the Boolean expression to logic gates
     - May need to map to AND/OR/NOT gates or to NAND or NOR only
     - May need to use Boolean alegbra or "Karnaugh maps" (see later) to obtain the simplest mapping to our target types of logic gate

A first example of logic circuit design (XOR)

- Consider a *stair-hall lighting circuit* – a light over the stairs is controlled by a switch **H** in the hall, and a switch **S** on the stairs
  - We want to be able to switch the light on at the bottom, and off again at the top—and vice versa
  - So, we want the light to be on if **S** is *up* and **H** is *down*, **or** if **S** is *down* and **H** is *up*

	S	H	Light
	0	0	0
S'H	0	1	1
SH'	1	0	1
	1	1	0

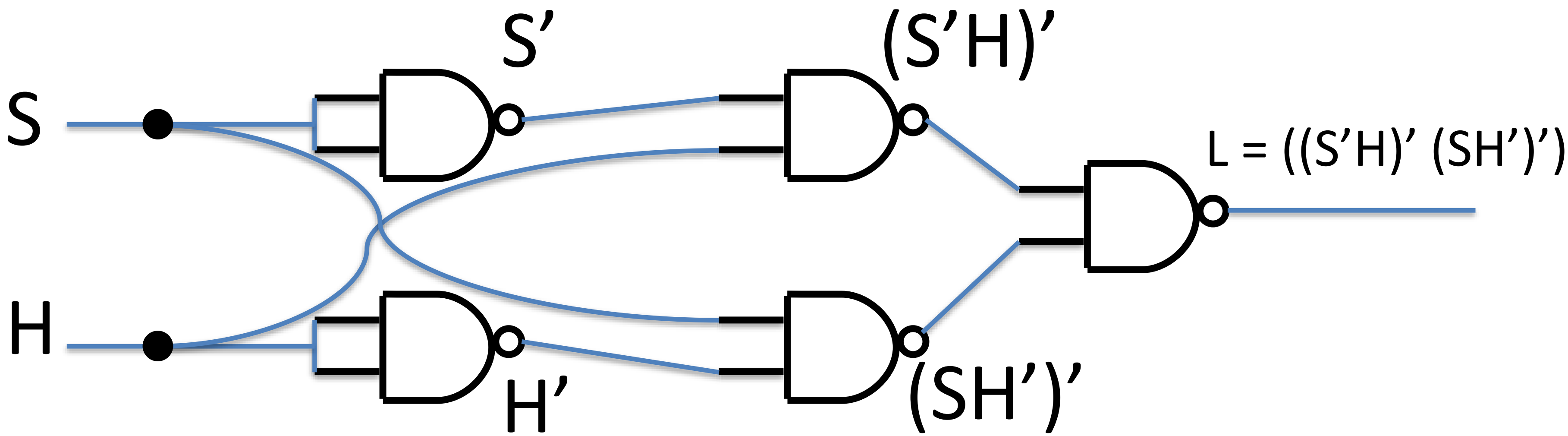
Example *contd*.

- So, L = S'H + SH' [*in **sum-of-products** form*]

- Let's implement this using NAND gates…

- Apply de Morgan's law
  - Let X = S'H and let Y = SH' So we have X + Y
  - By de Morgan's law, X + Y = (X' Y')'
  - Expand to ((S'H)' (SH')')' So L = ((S'H)' (SH')')'
  - This is now in the required "inverted AND" form…

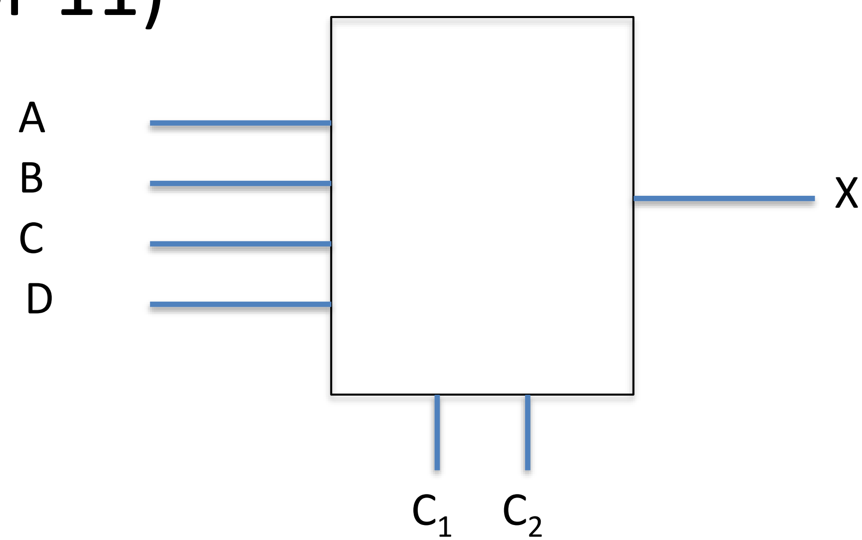
Example *contd*.



L = ((S'H)' (SH')')'

A second example: design a 4-way multiplexer

- We want a multiplexer that “lets though” one of four inputs A, B, C or D, depending on the values of control inputs C_1 and C_2 (i.e. 4 control possibilities: 00, 01, 10 or 11)
  - C_1=0 C_2=0 makes X=**A**
  - C_1=0 C_2=1 makes X=**B**
  - C_1=1 C_2=0 makes X=**C**
  - C_1=1 C_2=1 makes X=**D**



Multiplexer example *cont*.

- Write out the truth table and derive the sum-of-products form:

$$X = (AC_1'C_2') + (BC_1'C_2) + (CC_1C_2') + (DC_1C_2)$$

(each term is taken from an "X=1" row)

C_1	C_2	A	B	C	D	X
0	0	1	**X**	**X**	**X**	1
0	0	0	**X**	**X**	**X**	0
0	1	**X**	1	**X**	**X**	1
0	1	**X**	0	**X**	**X**	0
1	0	**X**	**X**	1	**X**	1
1	0	**X**	**X**	0	**X**	0
1	1	**X**	**X**	**X**	1	1
1	1	**X**	**X**	**X**	0	0

N.B., an X in a truth table means either 0 or 1 – i.e. "don't care"; this is useful in reducing the number of rows we have to consider!

Example *cont.*

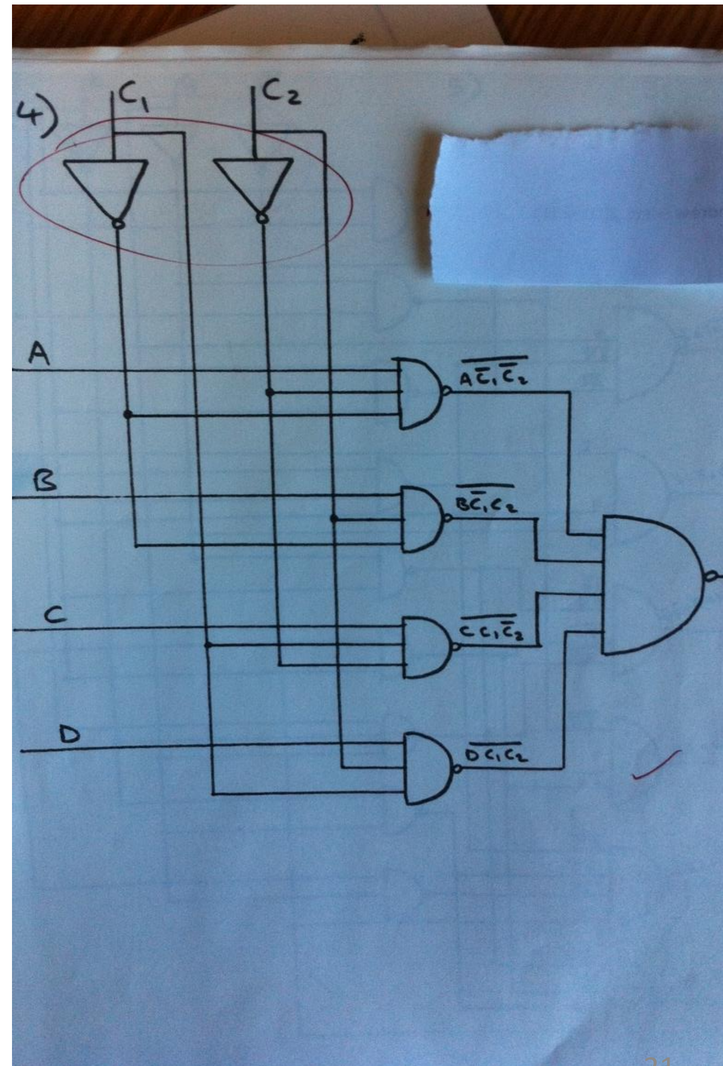
- Apply de Morgan's law to get this into "inverted AND" form

$$AC_1'C_2' + BC_1'C_2 + CC_1C_2' + DC_1C_2$$

$$=$$

$$(\ (AC_1'C_2')'\ \ (BC_1'C_2)'\ \ (CC_1C_2')'\ \ (DC_1C_2)'\ )'$$

- Now we can map directly to 3-input NANDs…



Using Karnaugh maps (K-map) to minimise Boolean logic functions

- Our examples so far have mapped "conveniently" to NAND gates; we have just needed de Morgan's law to translate to "inverted AND" form
  - But it's not always so straightforward in practice - using a truth table directly will often lead to a more complex implementation than necessary

- We can often *minimise* a Boolean function to produce a functionally equivalent, but simpler, implementation
- Karnaugh maps (K-map) offer an easy way to do this…

What is a Karnaugh map (K-map)?

- A grid in which each square represents one possible combination of inputs (cf. truth table row)
- Columns/rows are ordered so that only one input "changes" from col-to-col, and from row-to-row

(Also: note that Karnaugh maps "wrap" left-to-right and top-to-bottom)

	A	**A'**
B		
B'		

2-input map

	AB	**A'B**	**A'B'**	**AB'**
C				
C'				

3-input map

	AB	**A'B**	**A'B'**	**AB'**
CD				
C'D				
C'D'				
CD'				

4-input map

Using a Karnaugh map (K-map)

1. Pick a template with the required number of inputs, and put a 1 in any square for which we want an output of 1
2. Look for *rectangular groups* of 1s
   - Groups must contain 2 or 4 or 8 … (2^n) cells
   - Groups may overlap, and may wrap around the edges
   - The *larger* the groups, and the *fewer* the groups, the better

Result: for each group simply list the "unchanged" terms and OR them together ("changed" ones "cancel")

A′BCD + A′B′CD +
A′BC′D + A′B′C′D + AB′C′D +
AB′C′D′ +
ABCD′ + A′BCD′ + A′B′CD′ + AB′CD′

= A′D + AB′C′ + CD′

	AB	A'B	A'B'	AB'
CD		1	1	
C'D		1	1	1
C'D'				1
CD'	1	1	1	1

Karnaugh map example

- Implement a Decoder function that detects the following inputs: 0, 1, 2, 4 and 5 (assume 3-bit binary)
- Here's the truth table:

Decimal	Binary			Output
	A	B	C	
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	0

Identify the sum-of-products expression

- F = A’B’C’ + A’B’C + A’BC’ + AB’C’ + AB’C

Decimal	Binary			Output	Term
	A	B	C		
0	0	0	0	1	A’B’C’
1	0	0	1	1	A’B’C
2	0	1	0	1	A’BC’
3	0	1	1	0	
4	1	0	0	1	AB’C’
5	1	0	1	1	AB’C
6	1	1	0	0	
7	1	1	1	0	

Now, enter these five terms into a 3-input Karnaugh map template…

- **F = A'B'C' + A'B'C + A'BC' + AB'C' + AB'C**

	AB	A'B	A'B'	AB'
C	0	0	1	1
C'	0	1	1	1

Find the groups

- F = A’B’C’ + A’B’C + A’BC’ + AB’C’ + AB’C

	AB	A’B	A’B’	AB’
C			1	1
C’		1	1	1

The **larger** the groups the better
The **fewer** the groups the better…
(doesn’t matter if groups overlap)

Derive the result…

- Look for the “unchanged” variables in each group

<u>B’ is unchanged</u> (“A” cancels: appears as both A and A’; “C” cancels: appears as both C and C’)

	AB	A’B	A’B’	AB’
C			1	1
C’		1	1	1

<u>A’C’ is unchanged</u> (“B” cancels: appears as both B and B’)

Result

- Write down and OR together the“unchanged” variables…
  - **B’** was unchanged
  - **A’C’** was unchanged

- Result is therefore, **F = B’ + A’C’**

- (Can you derive an implementation using NAND gates? hint: apply de Morgan’s law to our result…)
- (Can you prove that A’B’C’ + A’B’C + A’BC’ + AB’C’ + AB’C = B’ + A’C’ using perfect induction?)

Be careful not to miss "wrap around" possibilities

- Remember that groups may "wrap around"
- So, in each of the following examples we have a single group of four cells

	AB	A'B	A'B'	AB'
CD				
C'D	1			1
C'D'	1			1
CD'				

A and C' are "unchanged"

	AB	A'B	A'B'	AB'
CD	1			1
C'D				
C'D'				
CD'	1			1

A and C are "unchanged"

Let’s do the same using Boolean algebra

- **F = A’B’C’ + A’B’C + A’BC’ + AB’C’ + AB’C**
  - Can use ***idempotence*** to expand:

    = **A’B’C’** + A’B’C + A’BC’ + **A’B’C’** + AB’C’ + AB’C
  - Can then use *distributivity* to combine “similar” pairs of terms:

    = A’B’**(C’ + C)** + A’**(B + B’)**C’ + AB’**(C’ + C)**
  - Can then use *complementarity* and then *identity-2* to simplify:

    = A’B’ + A’C’ + AB’ [X+X’=1 and then 1X=X]
  - Can then use *commutativity* to rearrange:

    = A’B’ + AB’ + A’C’
  - Can then use *distributivity* (again):

    = **(A’ + A)**B’ + A’C’
  - Can then use *complementarity* and then *identity-2* (again) to simplify:

    = B’ + A’C’

Do you prefer algebra or Karnaugh maps? :-)

Why do Karnaugh maps “work”?

- A Karnaugh map is simply a visual representation of a logical expression in sum-of-products form
- As we just saw, we can often simplify a sum-of-products expression like this:

  AB + AB’ => A(B + B’) => A

  [via distributivity=>complementarity=>identity-2]
- This is what we informally called « cancelling »
- This is essentially what a Karnaugh map does – e.g., the fact that the red group below includes both B and B’ allows the B variable to be cancelled from the expression

AB + AB’ =

	A	**A’**
B	**1**	
B’	**1**	

= A

Summary

- We know the four basic Boolean operators, and the corresponding logic gates
- We understand truth tables
- We appreciate the universality of NAND and NOR
- We understand the laws of Boolean algebra
- We promise to remember at least some of them (especially, de Morgan’s law)!
- We know how to go through the following process
  - a logic function specification → a truth table → a “sum of products” logic expression → a logic circuit
- We know how to minimise logic expressions using Karnaugh maps