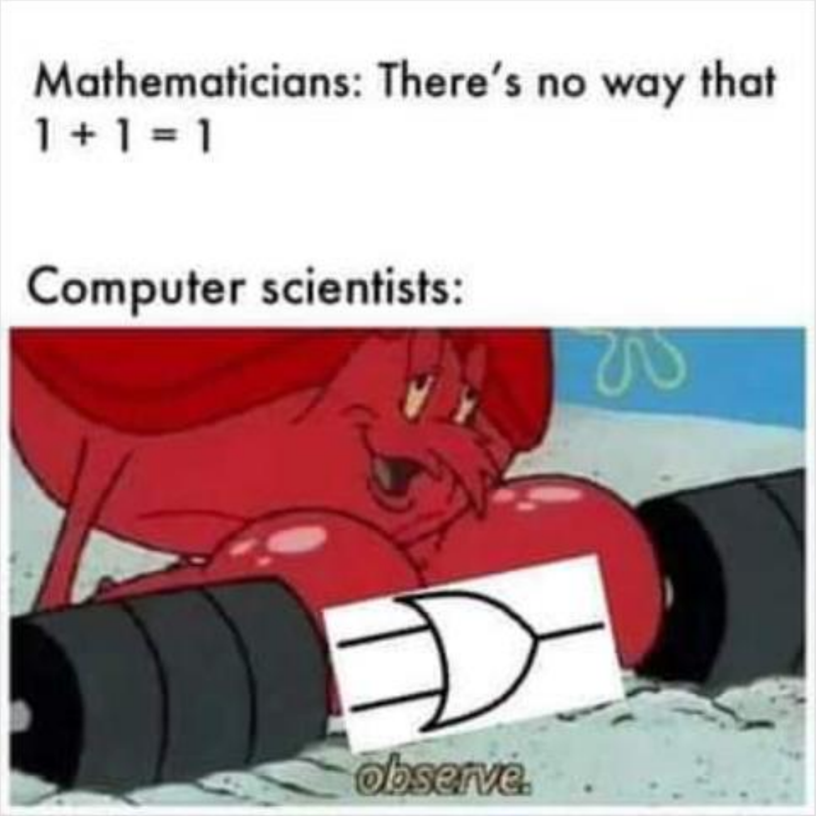
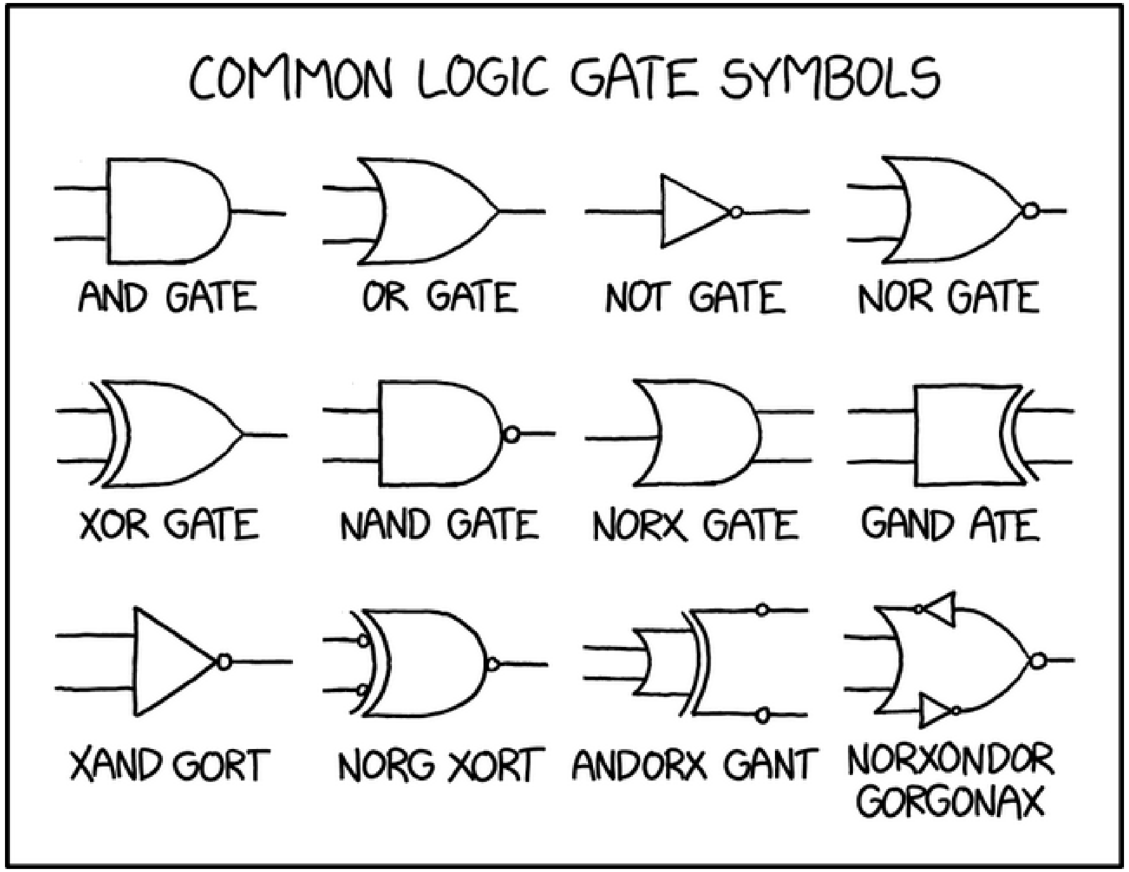
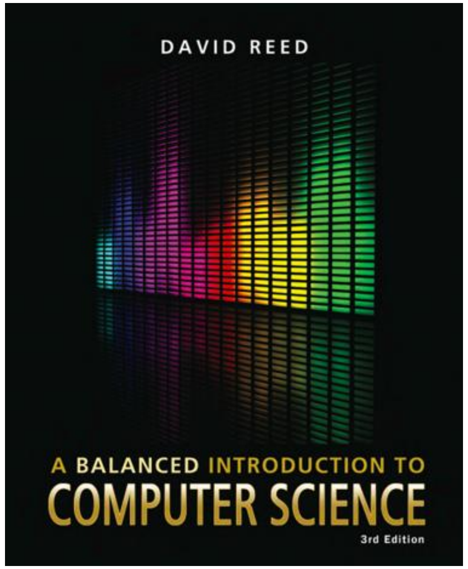
1. Computer science, Boolean algebra and logic gates





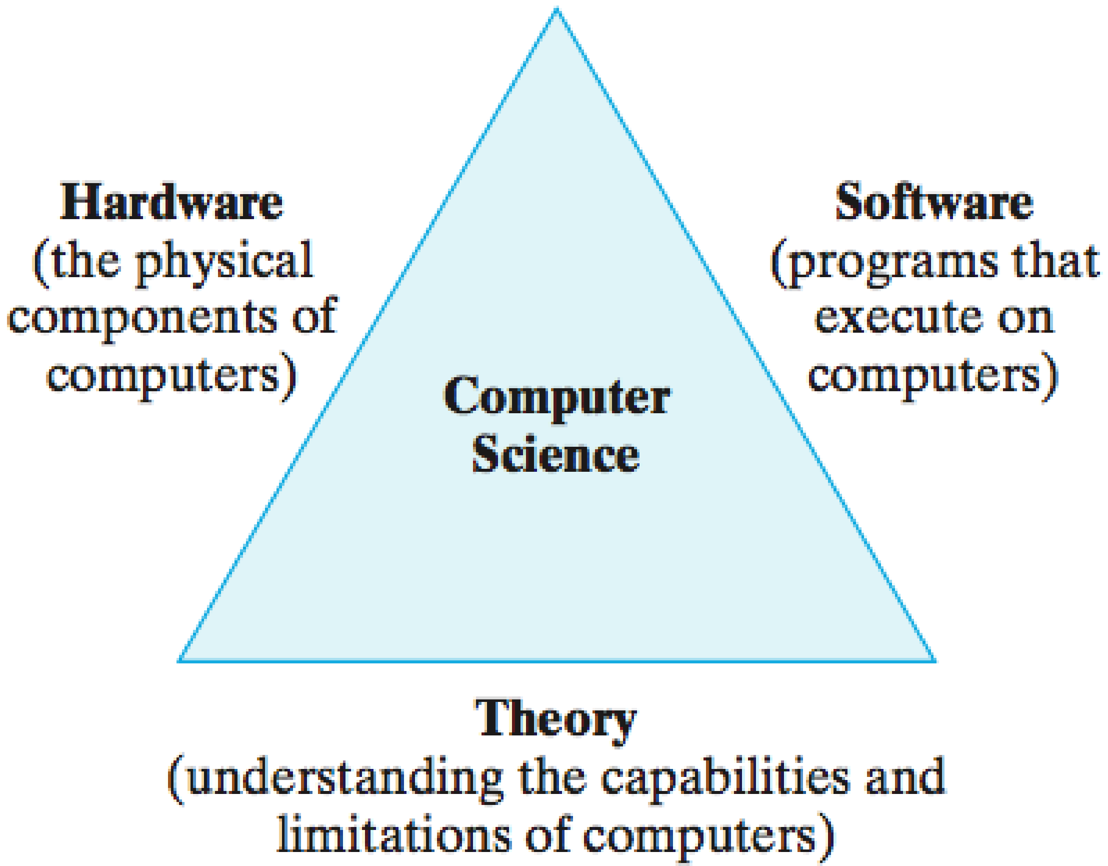
What is “computer science”?



- Computer science is very much different from all natural sciences
  - Some authors use the term “artificial science” for distinction
- It is the study of computation, which deals with problem solving:
  - Design and analysis of algorithms
  - Formalization of algorithms to programs
  - Development of computational devices for executing programs
  - Theoretical study of the power and limitations of computing
- Every other science aims to understand how the world works, and then model and predict its behavior
- Computer scientists, on the other hand, work in a universe that they have created by themselves; they try to create abstractions of real-world problems that can be understood and solved by computers

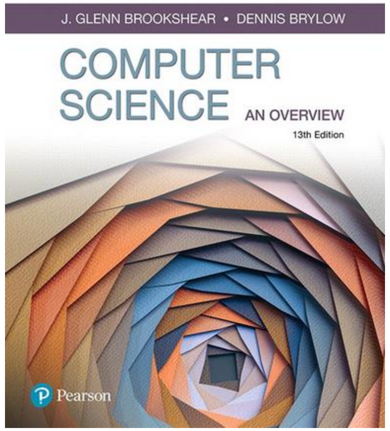
Computer science themes

- Computer science has three themes that define the discipline:
  - Hardware
  - Software
  - Theory
- Analogy: an athlete performing his/her sport
  - Hardware = sport-specific equipment (which enables the task)
  - Software = the athlete (who is performing the task)
  - Theory = trainers (who constantly think of ways to improve the result of the task)


Hardware
(the physical components of computers)
Software
(programs that execute on computers)
Computer Science
Theory
(understanding the capabilities and limitations of computers)

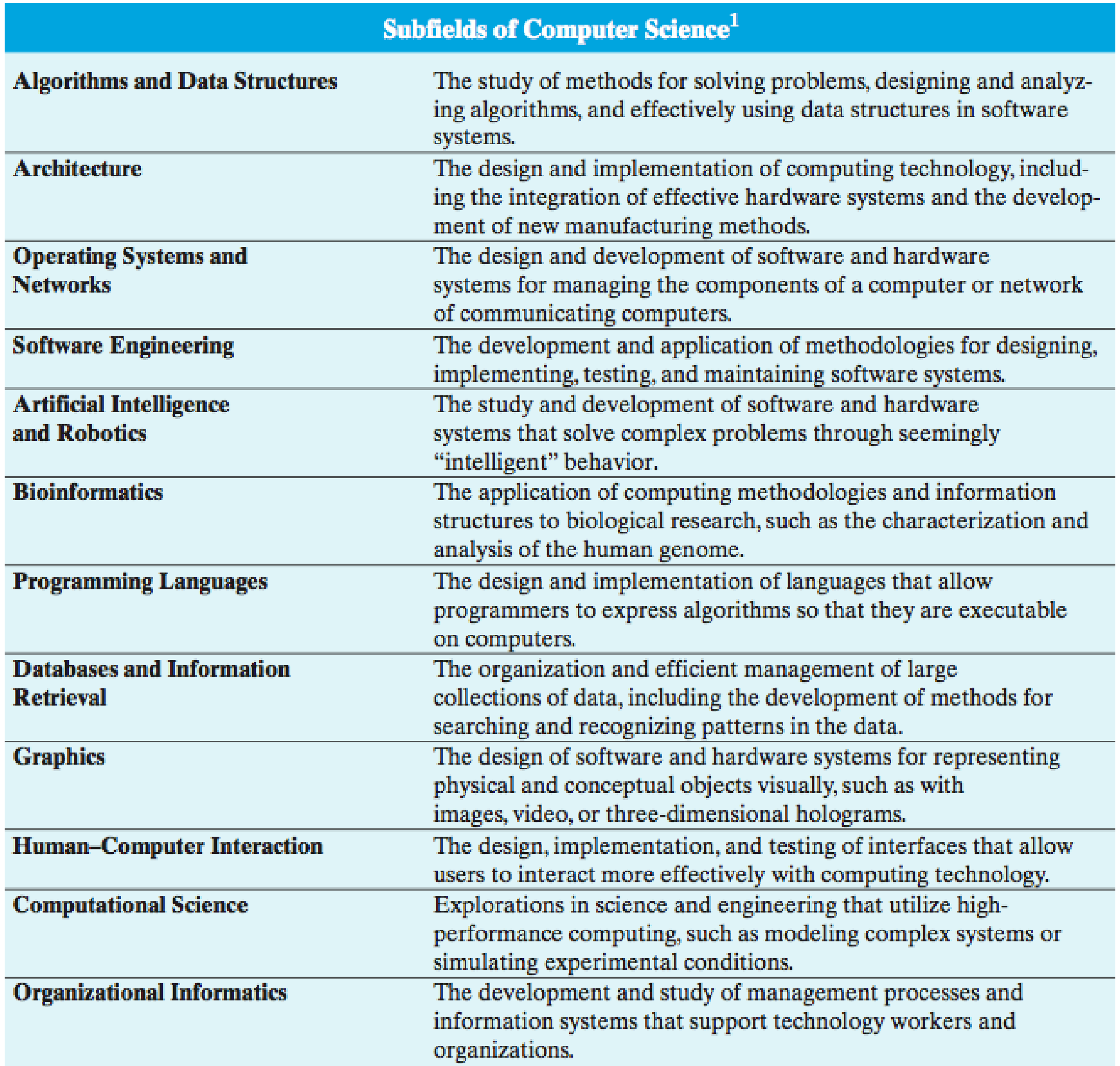
The “Seven Big Ideas of Computer Science” (Brookshear-Brylow)

- Algorithms
  - Step-by-step directions on how to perform a task; the origin of Computer Science
- Abstraction
  - Different levels of abstraction allow us to handle complex processes via abstract tools
- Creativity
  - A human is creative, a machine is not – it’s just performing the task it’s been given
- Data
  - How computers store known data, approximate unknown data and handle data errors?
- Programming
  - Translating human intentions into executable computer algorithms
- Internet
  - Connecting computers and electronic devices; privacy and security issues
- Impact
  - Influence of CS on the society (societal, ethical and legal)



Subfields of CS

- Computer science can be divided to several subfields
  - This division according to P. Denning already in 2000
  - Nowadays maybe even more?
- Many of these subfields are follow-up courses on their own
- This course tries to act as a primer for further studies, so these subfields are not considered very thoroughly



Subfields of Computer Science[1]	
Algorithms and Data Structures	The study of methods for solving problems, designing and analyzing algorithms, and effectively using data structures in software systems.
Architecture	The design and implementation of computing technology, including the integration of effective hardware systems and the development of new manufacturing methods.
Operating Systems and Networks	The design and development of software and hardware systems for managing the components of a computer or network of communicating computers.
Software Engineering	The development and application of methodologies for designing, implementing, testing, and maintaining software systems.
Artificial Intelligence and Robotics	The study and development of software and hardware systems that solve complex problems through seemingly "intelligent" behavior.
Bioinformatics	The application of computing methodologies and information structures to biological research, such as the characterization and analysis of the human genome.
Programming Languages	The design and implementation of languages that allow programmers to express algorithms so that they are executable on computers.
Databases and Information Retrieval	The organization and efficient management of large collections of data, including the development of methods for searching and recognizing patterns in the data.
Graphics	The design of software and hardware systems for representing physical and conceptual objects visually, such as with images, video, or three-dimensional holograms.
Human–Computer Interaction	The design, implementation, and testing of interfaces that allow users to interact more effectively with computing technology.
Computational Science	Explorations in science and engineering that utilize high-performance computing, such as modeling complex systems or simulating experimental conditions.
Organizational Informatics	The development and study of management processes and information systems that support technology workers and organizations.

Boolean algebra

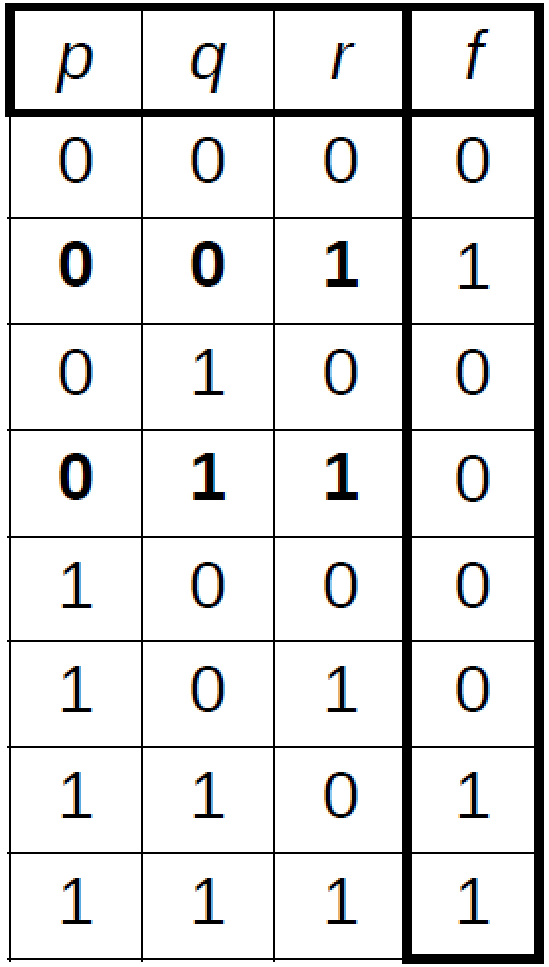
- Boolean algebra is an algebra where variables can only have two possible values: TRUE or FALSE – or 1 and 0, respectively
- Also, the number of possible operations that we can perform is limited to three:
  - Conjunction (“AND”, symbol ∧)
  - Disjuction (”OR”, symbol ∨)
  - Negation (”NOT”, symbol ¬ or overbar)
- Boolean expressions can be formed just like propositions in propositional logic
  - Propositional logic should be familiar from FoIP course
  - Propositions can be simplified to Boolean expressions by replacing ”arrows” (conditionals, biconditionals, etc.) with appropriate Booleans

Boolean laws

- Boolean algebra follows its own laws (which quite well match the simplification rules of propositional logic, presented earlier in FoIP course)
  - Commutativity: $x \wedge y = y \wedge x$ $x \vee y = y \vee x$
  - Distributivity: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
  - Complementation: $x \wedge \bar{x} = 0$ $x \vee \bar{x} = 1$
  - Identity: $x \wedge 1 = x$ $x \vee 0 = x$
  - Idempotence: $x \wedge x = x$ $x \vee x = x$
  - Associativity: $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ $(x \vee y) \vee z = x \vee (y \vee z)$
  - Absorption: $x \wedge (x \vee y) = x$ $x \vee (x \wedge y) = x$
  - De Morgan laws: $\overline{x \wedge y} = \bar{x} \vee \bar{y}$ $\overline{x \vee y} = \bar{x} \wedge \bar{y}$
  - Annihilation and double negation: $x \wedge 0 = 0$ $x \vee 1 = 1$ $\bar{\bar{x}} = x$

Boolean functions

- Every Boolean expression represents some Boolean function
  - When variables (or arguments) are given values, the Boolean function returns a TRUE/FALSE value
- Boolean functions can be displayed as truth tables:
  - Rows correspond to all possible combinations of truth values for the variables
  - Columns for each variable (here *p*, *q* and *r*) and a column for the output value of the function (here denoted as *f*)
- Size of the truth table depends on the number of variables
  - If a Boolean function has *k* variables, the truth table has 2^k rows
  - Therefore, the truth table contains all possible variable combinations
  - Tip: Write truth tables in binary order (easier to check)



p	*q*	*r*	*f*
0	0	0	0
0	**0**	**1**	1
0	1	0	0
0	**1**	**1**	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Boolean functions

- While truth tables are a nicely “complete” way to represent Boolean functions, their exponentially growing size forces us to find other means to evaluate them
  - 10 variables → $2^{10} = 1024$ rows
- For evaluation purposes, it is enough to find a Boolean expression that returns exactly the same output as the original Boolean function
- For each Boolean function, there exists an infinite amount of such expressions
  - How could we choose the simplest one?
  - Simplification using Boolean laws, Karnaugh maps, …
- Finding out the simplest possible expression is, in general, a quite difficult task
  - Luckily, we don’t usually need THE simplest one (if such even exists)
- Good baseline for simplification AND also for design of logical circuits (which perform Boolean operations in reality) is to present the expression in normal form

Literals and elementary disjunctions

- Expressions which are either single propositional variables or their negations are called *literals*
- A Boolean expression is said to be an *elementary disjunction* if it has the form

$$A_1 \vee A_2 \vee \cdots \vee A_n$$

where

- Every A_i is a literal
- Every variable appears only in one A_i
- Literals have (some) order

Example:

$p \vee q \vee \bar{r}$ is an elementary disjunction

$p \vee \bar{p} \vee s$ is not (p present in two literals!)

Conjunctive normal form

- A Boolean expression is in *Conjunctive normal form* (CNF, also known as "Product of Sums" = POS), if it has the form

$$B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

where

- Every B_i is an elementary disjunction
- None of the B_i's are the same
- Elementary disjunctions have (some) order

Example:

$(p \vee q \vee \bar{r}) \wedge (\bar{q} \vee s)$ is a CNF

$(p \wedge (q \vee \bar{r})) \wedge (p \vee r)$ is not (B_1 is not an elementary disjunction!)

Disjunctive normal form

- A Boolean expression is in *Disjunctive normal form* (DNF; also known as “Sum of Products” = SOP), if it has the form

$$B_1 \vee B_2 \vee \cdots \vee B_n$$

where

  - Every B_i is an *elementary conjunction* $(A_1 \wedge A_2 \wedge \cdots \wedge A_n)$
  - None of the B_i’s are the same
  - Elementary conjunctions have (some) order

Example of a DNF:

$(p \wedge q) \vee (\bar{p} \wedge \bar{q} \wedge s)$

- Notice: the difference between CNF and DNF
  - CNF = operators between brackets are conjunctions
  - DNF = operators between brackets are disjunctions

$(p \vee q) \wedge (q \vee s)$ **CNF**

$(p \wedge q) \vee (q \wedge s)$ **DNF**

From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression
- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

EXAMPLE:

p	q	r	f
0	0	0	0
0	**0**	**1**	1
0	1	0	0
0	**1**	**1**	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

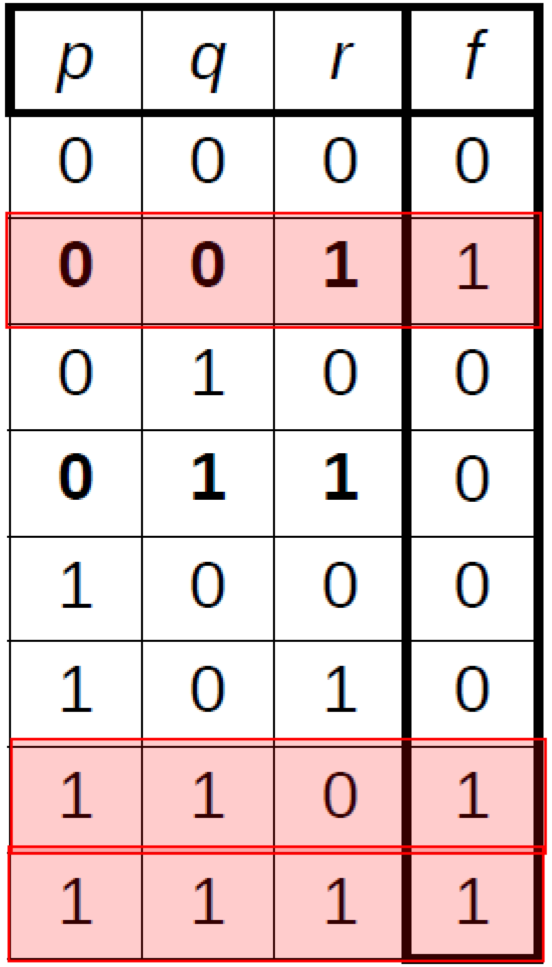
From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression

$DNF: (\bar{p} \wedge \bar{q} \wedge r) \vee (p \wedge q \wedge \bar{r}) \vee (p \wedge q \wedge r)$

- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

EXAMPLE:



p	*q*	*r*	*f*
0	0	0	0
0	**0**	**1**	1
0	1	0	0
0	**1**	**1**	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression
- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

EXAMPLE:

p	q	r	f
0	0	0	0
0	**0**	**1**	1
0	1	0	0
0	**1**	**1**	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$CNF: (p \vee q \vee r) \wedge (p \vee \bar{q} \vee r) \wedge (p \vee \bar{q} \vee \bar{r}) \wedge (\bar{p} \vee q \vee r) \wedge (\bar{p} \vee q \vee \bar{r})$

Shorthand notation

- As you probably noticed, the mathematical notation containing conjunctions and disjunctions is not very reader-friendly
- In applications it is common to use shorthand notation:
  - Conjunctions are represented as multiplication (with multiplication symbol omitted)
  - Disjunctions are represented as summation (+)
  - Negations are represented by overbars OR an apostrophe (‘)
- This results in expressions which don’t require a math editor!
- Example: DNF and CNF expressions from previous slides

$DNF: (\bar{p} \wedge \bar{q} \wedge r) \vee (p \wedge q \wedge \bar{r}) \vee (p \wedge q \wedge r)$ → p’q’r+pqr’+pqr

$CNF: (p \vee q \vee r) \wedge (p \vee \bar{q} \vee r) \wedge (p \vee \bar{q} \vee \bar{r}) \wedge (\bar{p} \vee q \vee r) \wedge (\bar{p} \vee q \vee \bar{r})$

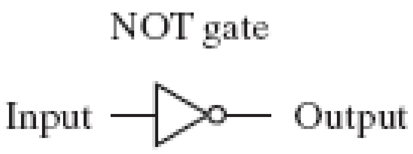
→ (p+q+r)(p+q’+r)(p+q’+r’)(p’+q+r)(p’+q+r’)

Semiconductors

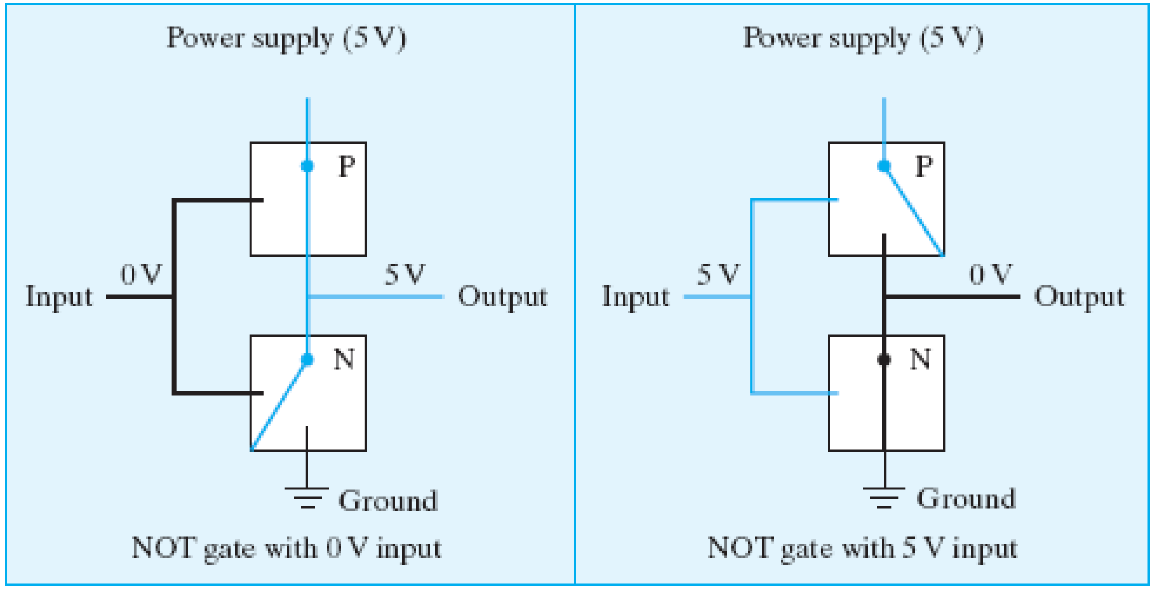
- Semiconductors are metals that can be manipulated to be either good or bad conductors of electricity
  - Most common semiconductor material is silicon
- This manipulation is done by adding impurities; the process is called “doping”
  - Negative doping = addition of atoms which have more valence electrons than base material
  - Positive doping = addition of atoms which have less valence electrons than base material
- Positively doped semiconductors are called p-type and negatively doped ones are called n-type
- Simple electrical components consist of these:
  - Diode = p-n-junction, that allows electricity to flow in only one direction
  - Transistor = two consecutive n-p-junctions, where electricity flow can be controlled via base voltage; can be used as a small and fast switch

Logic gates

- Propositional logic and Boolean algebra can be used to design digital circuits
  - Digital = two voltage levels; “low” and “high” – corresponding to 0 and 1 in Boolean
- Diodes and transistors can be used to build logic gates that execute the Boolean functions in real life
- For example a NOT gate (negation)
  - Notice the drawing symbol (small circle)!



input	NOT output
0	1
1	0

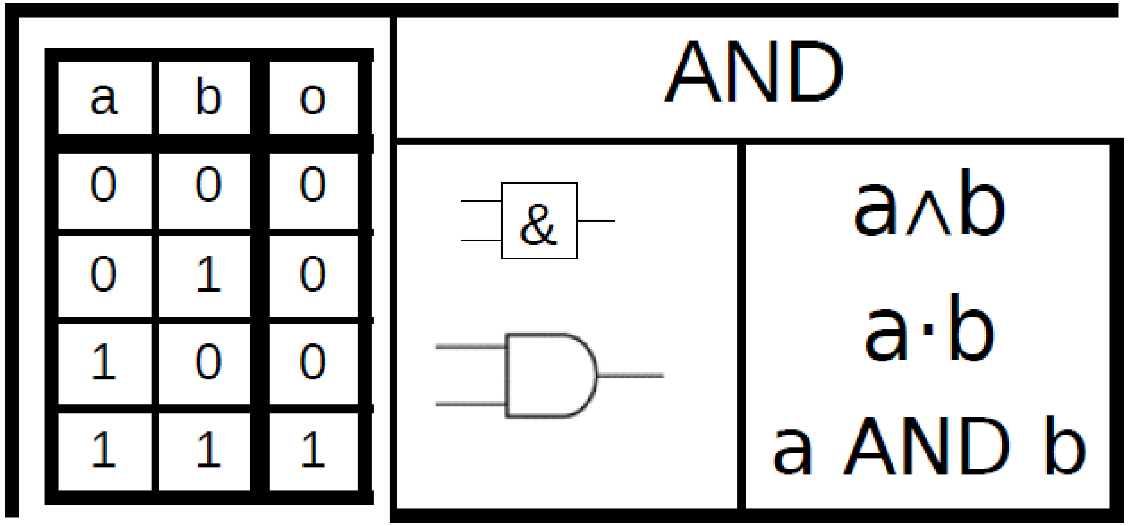


NOT gate with 0 V input

NOT gate with 5 V input

Logic gates

- In similar fashion, AND & OR gates:
  - Two possible drawing symbols (the lower ones are more commonly used)
  - Inputs (left) are a and b, o is the output (right)



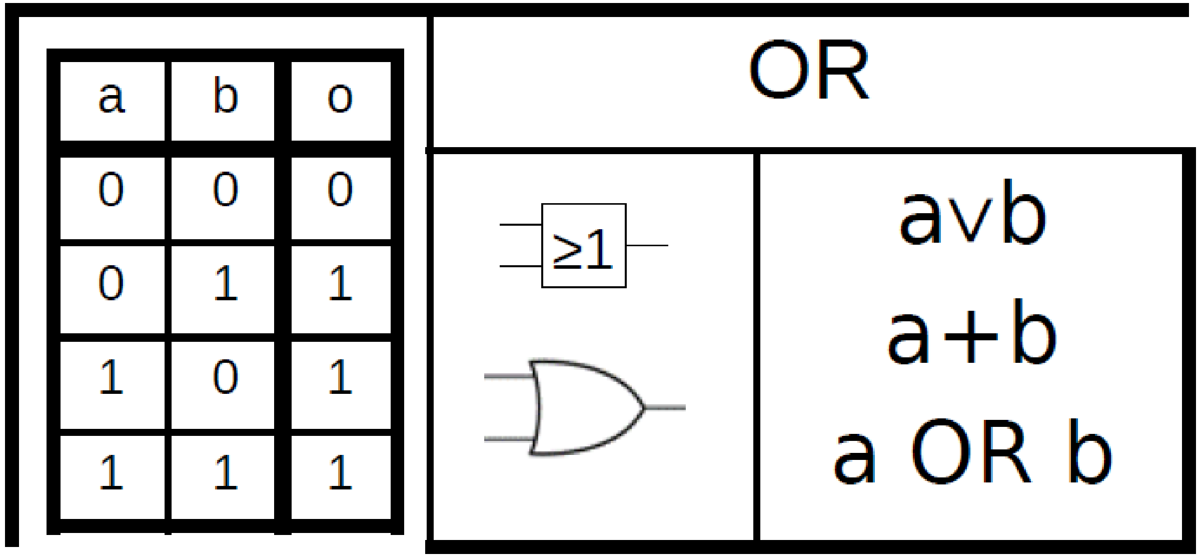
AND

a	b	o
0	0	0
0	1	0
1	0	0
1	1	1

$a \wedge b$

$a \cdot b$

a AND b



OR

a	b	o
0	0	0
0	1	1
1	0	1
1	1	1

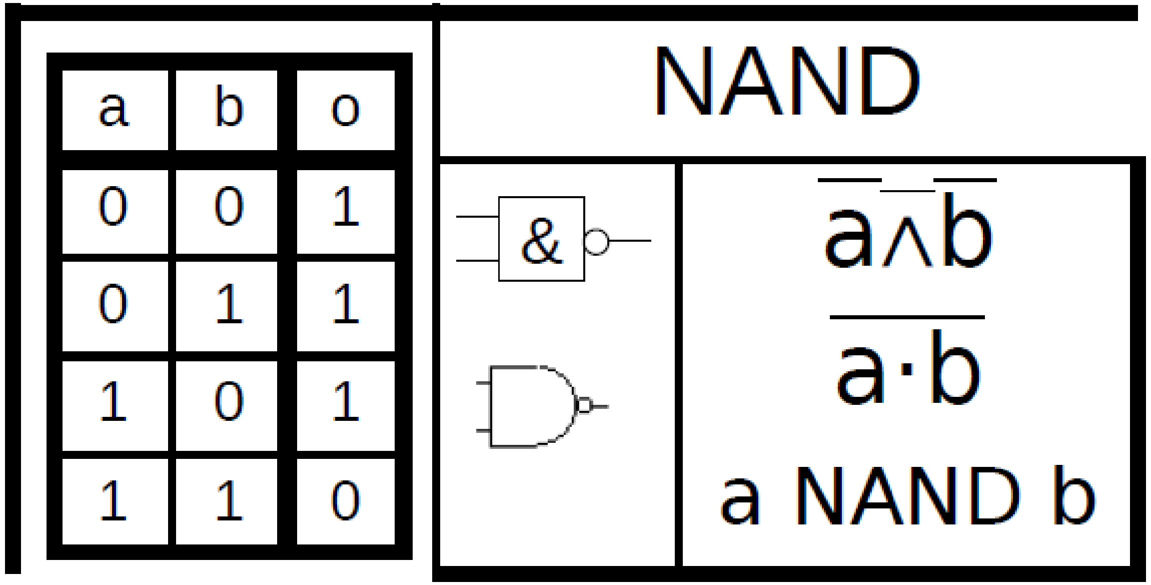
$a \vee b$

$a + b$

a OR b

Logic gates

- In addition to the three previous ones (which are basic Boolean gates), there are also additional gates which come in handy:
  - NAND = “Not AND”; returns 0 if both inputs are true and 1 otherwise
  - NOR = “Not OR”; returns 1 if both inputs are false and 0 otherwise



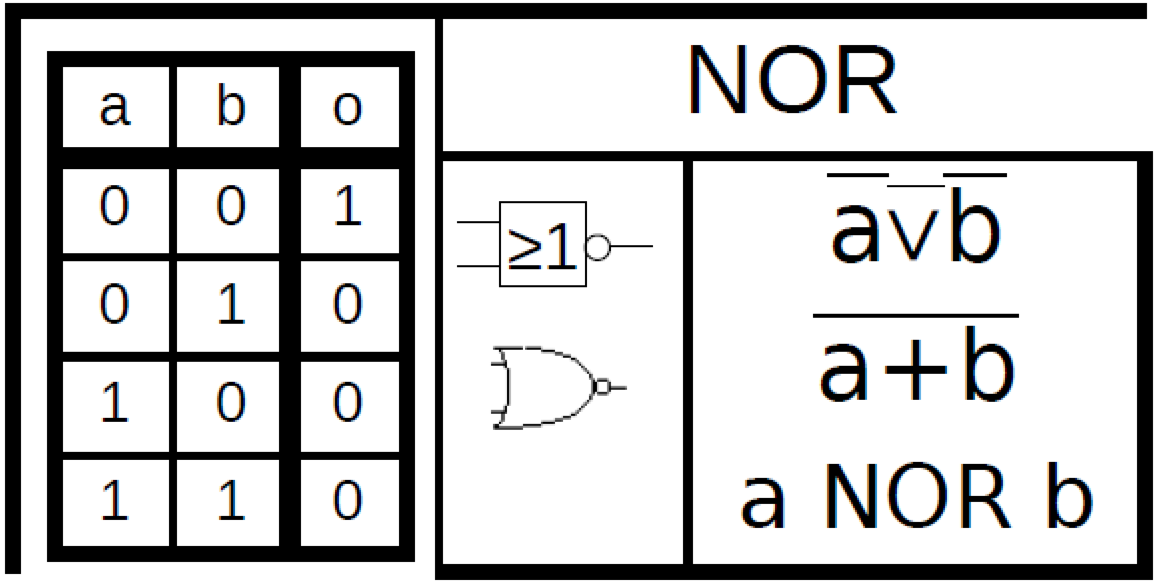
NAND

a	b	o
0	0	1
0	1	1
1	0	1
1	1	0

$\overline{a \wedge b}$

$\overline{a \cdot b}$

a NAND b



NOR

a	b	o
0	0	1
0	1	0
1	0	0
1	1	0

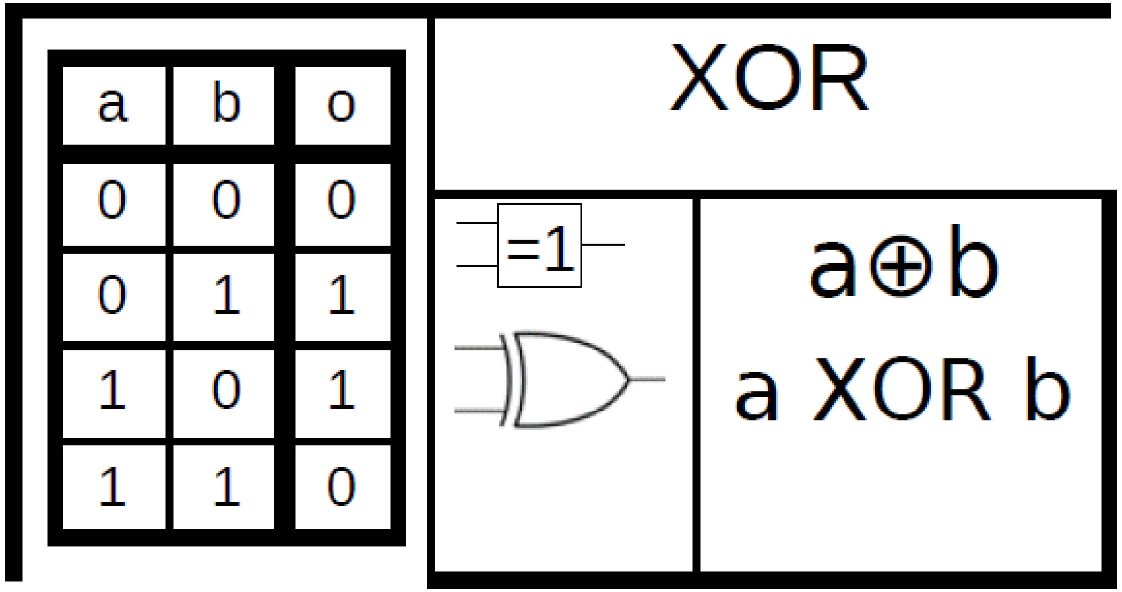
$\overline{a \vee b}$

$\overline{a+b}$

a NOR b

Logic gates

- Two more:
  - XOR = “Exclusive OR”; returns 0 if inputs are the same and 1 if they differ
  - EQV or XNOR = “Equivalent”; returns 1 if inputs are the same and 0 if they differ

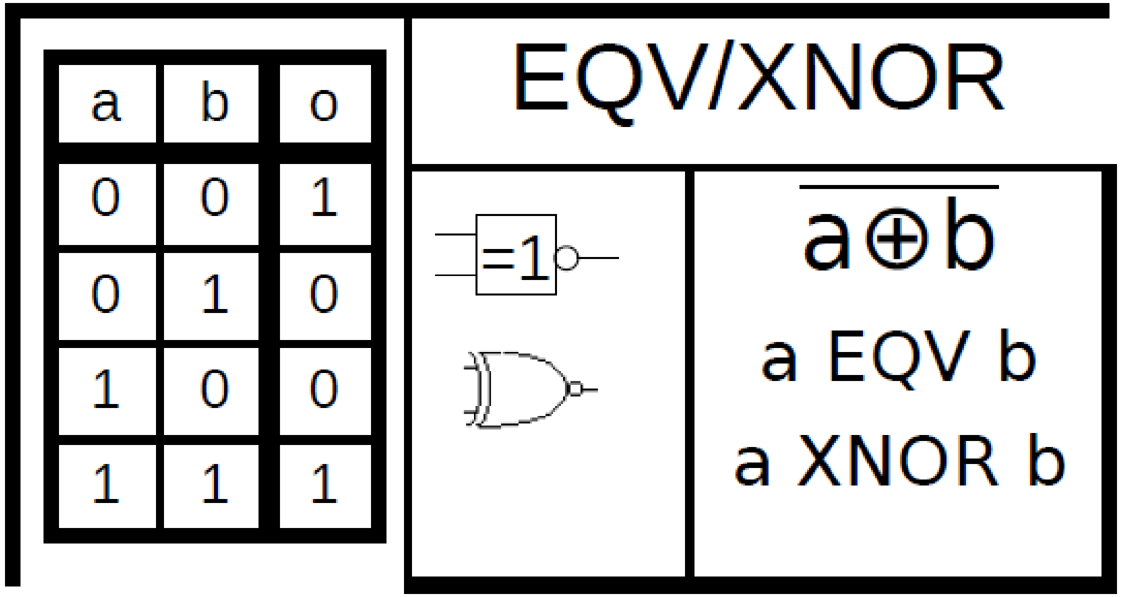


XOR

a	b	o
0	0	0
0	1	1
1	0	1
1	1	0

$a \oplus b$

a XOR b



EQV/XNOR

a	b	o
0	0	1
0	1	0
1	0	0
1	1	1

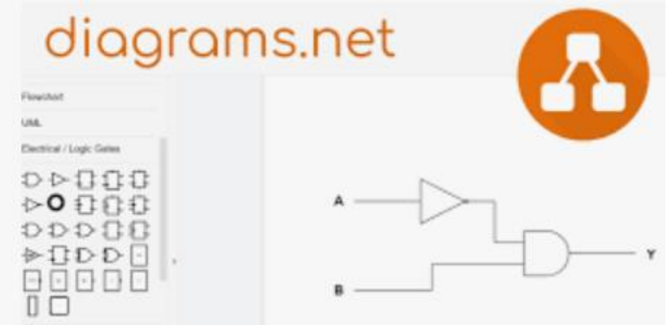
$\overline{a \oplus b}$

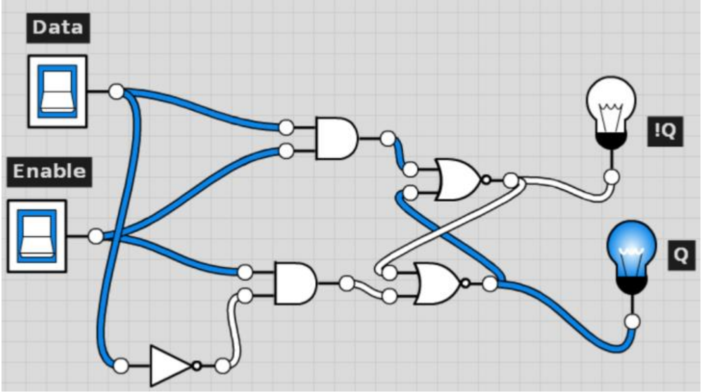
a EQV b

a XNOR b

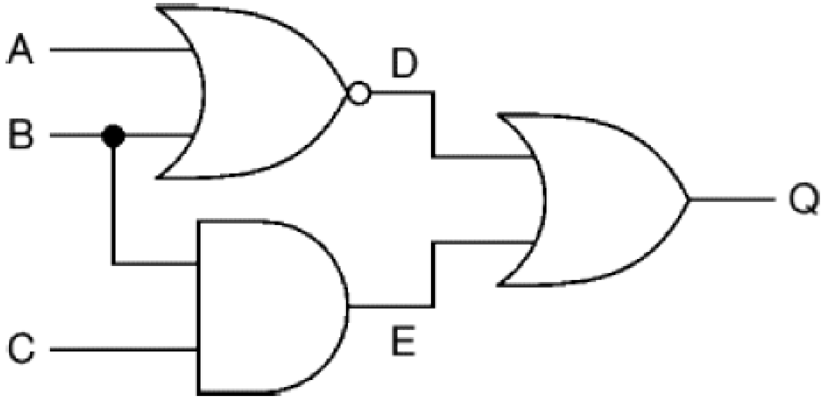
Logic circuits

- More advanced logic circuits can be constructed by combining several gates
- The circuit can be presented by
  - Drawing a logic diagram using previous symbols
  - Giving a Boolean function
- The purpose and action of the circuit can be shown by a truth table
  - Note! A truth table doesn't uniquely specify the circuit; same truth table can be achieved by various circuits!
- Tip: There are several online tools which can be used to sketch a logic circuit and test how it works
  - logic.ly
  - draw.io etc…



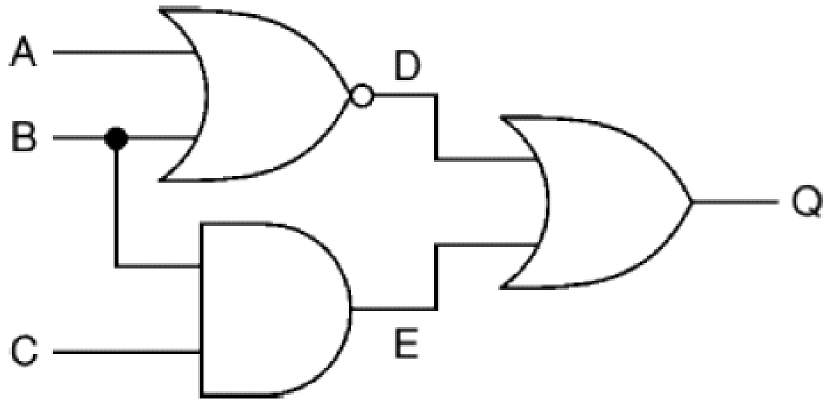


From logic diagram to truth table



From logic diagram to truth table

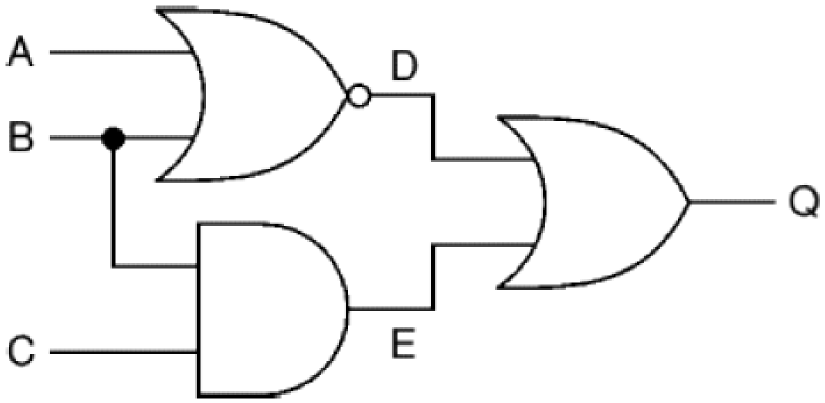
- 1) Decipher the gate symbols.



D = NOT (A OR B)
E = B AND C
Q = D OR E
= (NOT (A OR B)) OR (B AND C)

From logic diagram to truth table

- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.

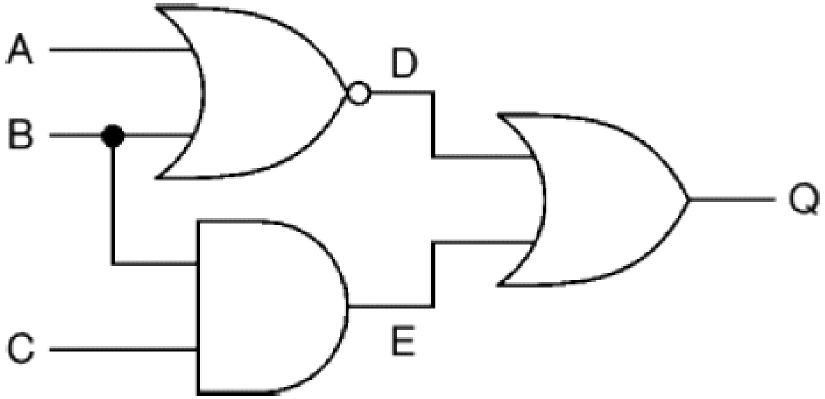


D = NOT (A OR B)
E = B AND C
Q = D OR E
= (NOT (A OR B)) OR (B AND C)

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

From logic diagram to truth table

- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.
- 3) Find out the output values layer by layer.

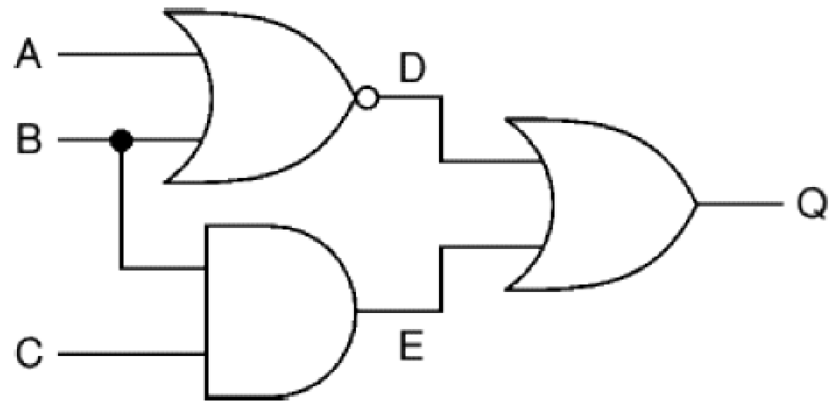


D = NOT (A OR B)
E = B AND C
Q = D OR E
= (NOT (A OR B)) OR (B AND C)

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0	1	0	
0	0	1	1	0	
0	1	0	0	0	
0	1	1	0	1	
1	0	0	0	0	
1	0	1	0	0	
1	1	0	0	0	
1	1	1	0	1	

From logic diagram to truth table

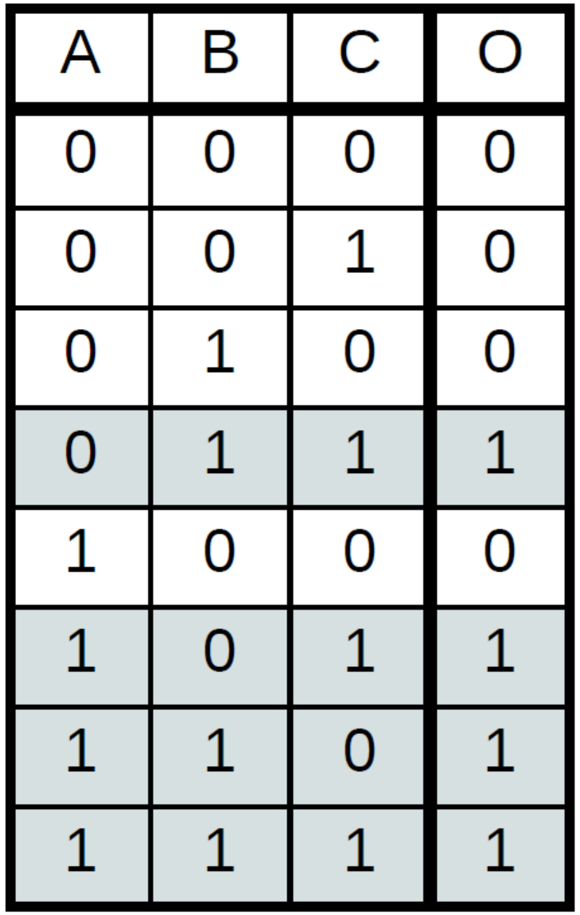
- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.
- 3) Find out the output values layer by layer.



D = NOT (A OR B)
E = B AND C
Q = D OR E
= (NOT (A OR B)) OR (B AND C)

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

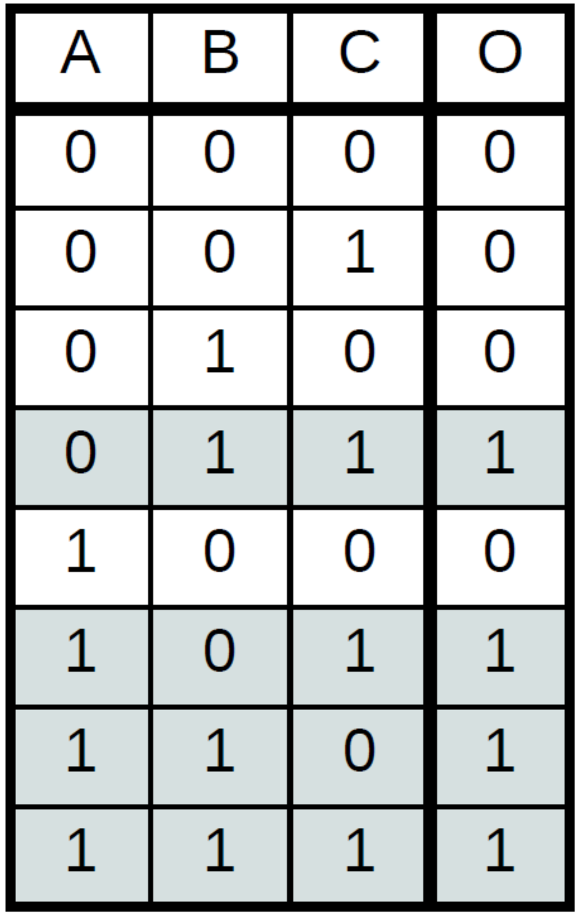
From truth table to circuit



A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF of DNF



A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\boldsymbol{DNF}:$$

$$\boldsymbol{O = A'BC + AB'C + ABC' + ABC}$$

From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF of DNF
- 2) Simplify the expression using Boolean laws (if possible)

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$\boldsymbol{DNF}$:

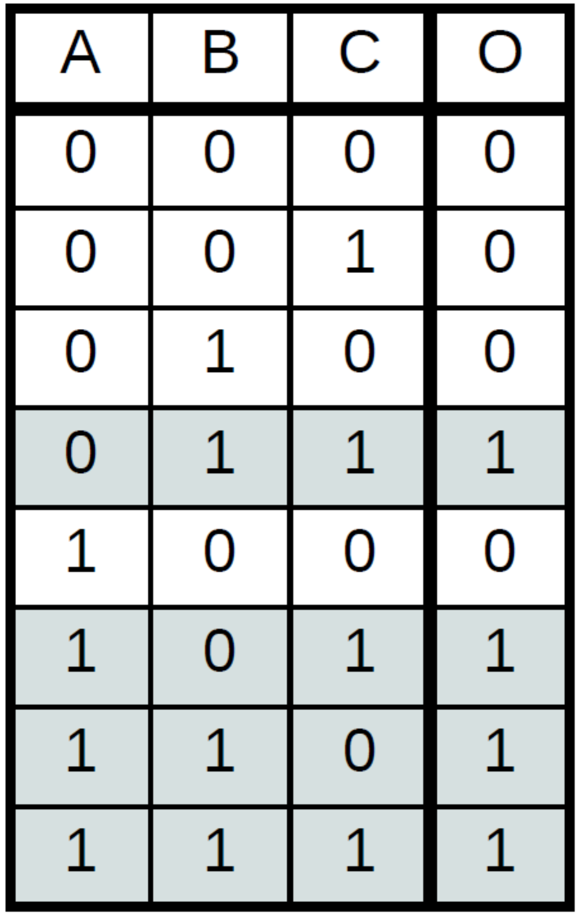
$$\boldsymbol{O = A'BC + AB'C + ABC' + ABC}$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$\boldsymbol{= (A'B + AB')C + AB(C' + C)}$$

From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF of DNF
- 2) Simplify the expression using Boolean laws (if possible)



A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\boldsymbol{DNF}:$$

$$\boldsymbol{O = A'BC + AB'C + ABC' + ABC}$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$= (\boldsymbol{A'B + AB'})\boldsymbol{C} + \boldsymbol{AB}(\boldsymbol{C' + C})$$

Use complementation law on the latter:

$$= (\boldsymbol{A'B + AB'})\boldsymbol{C + AB}$$

From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF of DNF
- 2) Simplify the expression using Boolean laws (if possible)
- 3) Draw the circuit using standard drawing symbols

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\boldsymbol{DNF}:$$

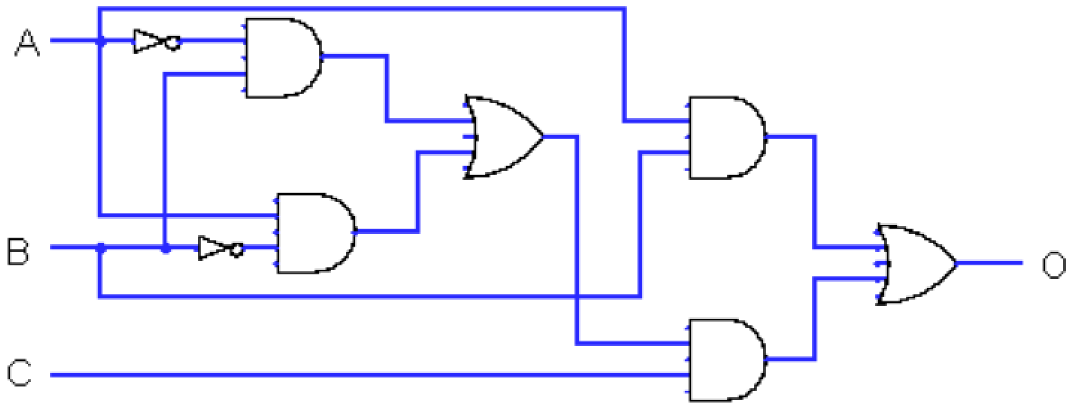
$$\boldsymbol{O = A'BC + AB'C + ABC' + ABC}$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$= (\boldsymbol{A'B + AB'})\boldsymbol{C} + \boldsymbol{AB}(\boldsymbol{C' + C})$$

Use complementation law on the latter:

$$= (\boldsymbol{A'B + AB'})\boldsymbol{C} + \boldsymbol{AB}$$

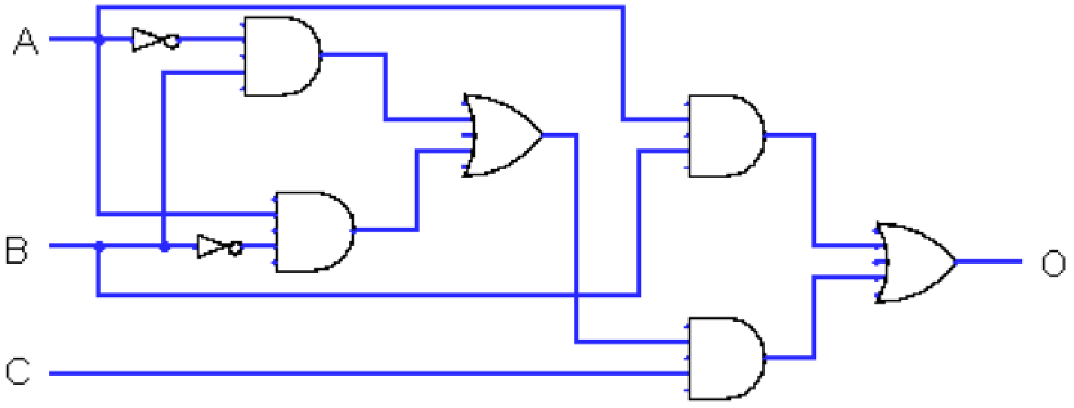


From truth table to circuit

- Tip: you can check your circuit by counting the operations

$$(A'B+AB')C+AB$$

  - 4 multiplications, 2 summations, 2 negations
- These must match the number of gates in your circuit!
  - 4 AND gates, 2 OR gates, 2 NOT gates
- If you want to use more advanced gates (NAND, NOR, XOR, EQV,…), this naturally doesn't work (because they're not Boolean operators)



Now it should be easy to notice why simplification is important: The original Boolean function in DNF was

$$A'BC+AB'C+ABC'+ABC$$

…this contains 8 multiplications, 3 summations, 3 negations -> in total 14 gates!
The simplified version contains only 8 gates.

Thank you for listening!