# Chapter 2 and 3: Logic Gates and Logic

## Chapter 2:

At a high level, computer components (such as central processing units or CPUs) are constructed from **digital circuits** which are **constructed from logic gates which are in turn ultimately constructed from transistors**.

See <https://www.cs.bu.edu/~best/courses/modules/Transistors2Gates/> for overview of transistors, gates and logic.

### Transistors and Switches

A **transistor** is eﬀectively **a digital switch which is used to either establish or break an electrical connection**, in much the same way that a light switch can either connect or disconnect a light bulb to or from household current.

Transistors as digital switch
Break in connection on left
Connection with no break on right

Diagrammatic representation of a digital switches. The left switch is “normally open” while the right switch is “normally closed.” Left switch is “normally open” and “pushing” the switch establishes the electrical connection, while the right switch is “normally closed” and “pushing” the switch breaks the electrical connection.

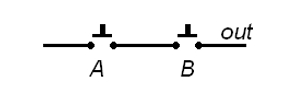
### Basic Logic Gates: AND, OR, NOT

Switches can be wired together to form basic logic gates which are used to construct circuits which can manipulate numbers. The basic logic gates are the AND, OR, and NOT gates.

#### AND Gate

An AND gate takes two inputs (switches) and is “on” (switched) so long as both switches have been “pushed”.

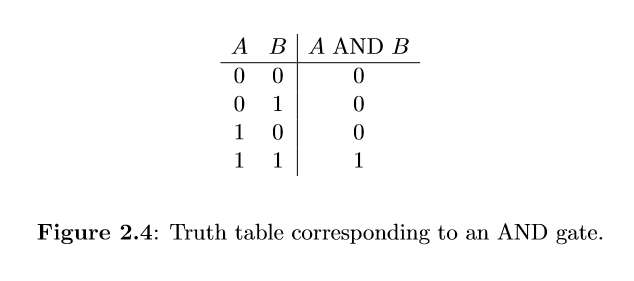
In terms of switches, an AND gate is represented diagrammatically as follows. In this diagram, A and B represent the two input switches, and a connection is established,



only if both switches are “pushed.” Actual CPUs constructed from circuits, logic gates, and ultimately transistors do not function physically like switches in that no transistor is actually ever “pushed.” Instead, **a “high” voltage (typically +5V) given as input to a transistor causes it to “close” and supply a “high” voltage to its output**; similarly, **a “low” voltage (typically 0V) given as input to a transistor causes it to remain “open” and supply no voltage (i.e., 0V) to its output**. Physically and logically, binary **1s and 0s are represented by these “high” and “low” voltages**, respectively. Given this representation, we can describe the action of an AND gate using a truth table. For example, the truth table corresponding to the possible actions of an AND gate are given below. Given two inputs (A and B) which can each take on two values (0 or 1), there are four possible input pairs to the AND gate.

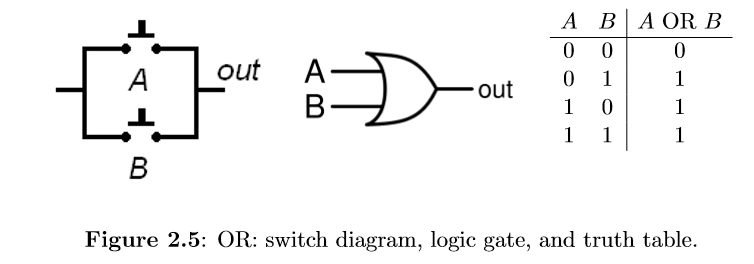
Logic gate AND with AB input and output
Figure 2.3

Each row in the truth table corresponds to one such input pair, and the corresponding output of the AND gate is also given. Note that the “A AND B” is 1 if and only if both A and B are 1; this corresponds to the logical idea that for a connection to be established, both switches must be “pushed.”



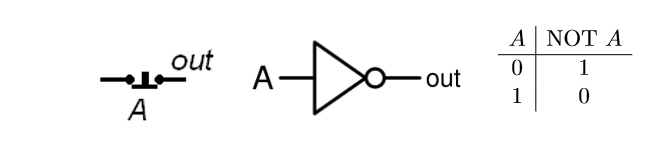
#### OR Gate

An OR gate takes two inputs and is “on” so long as at least one of the inputs is “on.” Note that **an OR gate is 1 (“on”) if and only if at least one of its inputs is 1**, and note how this is realized physically with switches.



#### NOT Gate

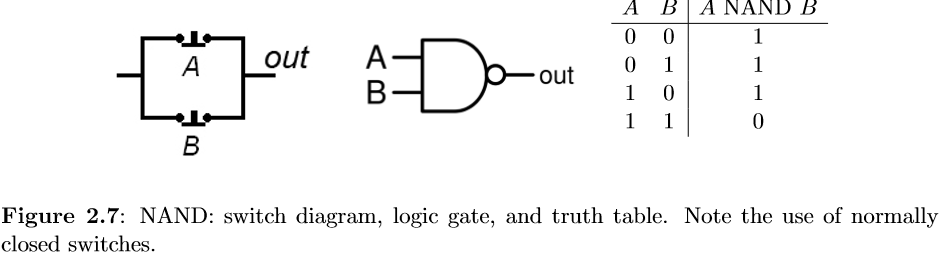
Unlike the AND and OR gates, **the NOT gate has only one input**, and **its output is simply the opposite of its input.** Note that in the switch diagram, the switch is of the “normally closed” variety; pushing the switch breaks the connection in this case.



#### NAND Gate

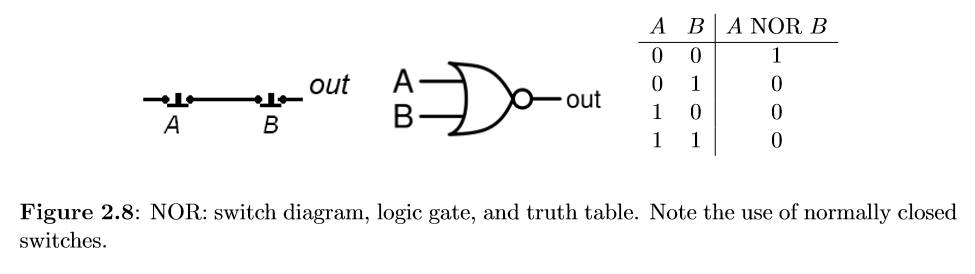
The **NAND gate is the opposite of an AND gate**: it is 1 (on) if and only if it is not the case that both of its inputs are 1. **A NAND gate can be constructed from an AND gate whose output is attached to a NOT gate**. The switch diagram, logic gate representation, and truth table for a NAND gate is given below. The NAND gate has two interesting properties:

* It is the **simplest logic gate to construct from common electrical components (transistors, resistors, wires, etc.)** or to fabricate as part of an integrated circuit.
* The NAND gate is **“logically complete”** in that every conceivable truth table, logic gate, or circuit can be constructed solely from NAND gates.



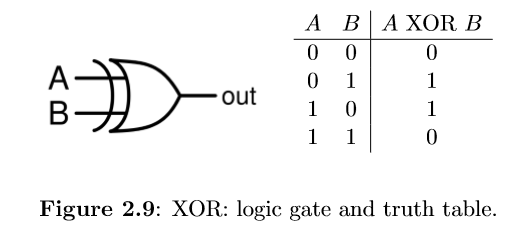
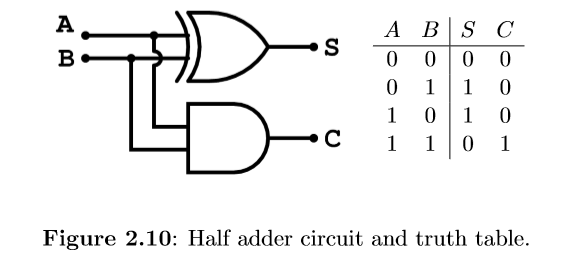
#### NOR Gate

**The NOR gate is the opposite of an OR gate**: it is 1 (on) if and only if it is not the case that at least one of its inputs 1. **A NOR gate can be constructed from an OR gate whose output is attached to a NOT gate.** The switch diagram, logic gate representation, and truth table for a NOR gate is given below.



#### XOR Gate

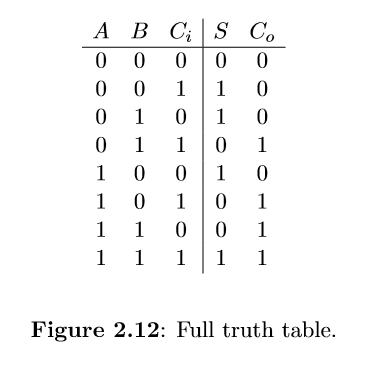
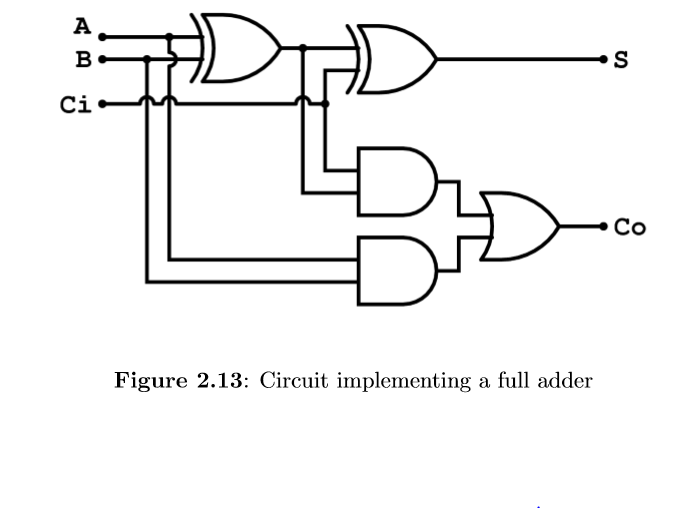
The XOR gate is the **“exclusive OR”** gate; it is 1 (on) if and only if one input is 1, but not both. The logic gate representation and truth table for a XOR gate is given below. **The XOR gate is very useful in implementing binary arithmetic.** Consider adding two binary digits: if both bits are 0, the sum is 0; if one of the bits is 0 and the other bit is 1, the sum is 1; and if both bits are 1, the sum is 2, or in binary, 10. **Note that the XOR gate gives the proper output of the least signiﬁcant bit in adding two bits, and further note that an AND gate gives the proper output of the most signiﬁcant bit (or carry) in adding two bits.** Such a simple circuit is called a half adder; see the ﬁgure below.

### Binary Arithmetic: Ripple Carry Adders

In order to perform **the addition of two binary numbers, one must in each column sum the corresponding bits from each input number together with any input carry bit, producing an output bit and possibly a carry bit.**

Letting A, B, and Ci denote the ﬁrst and second input bits and the input carry bit, and letting S and Co denote the output sum and carry bit, the following truth table shown in Figure 2.12 represents the required action for a circuit dealing with one column of binary addition; a circuit implementing this truth table is shown in Figure 2.13 Stringing together a series of full adders, one for each column binary addition, yields a ripple carry adder as shown in Figure 2.14.

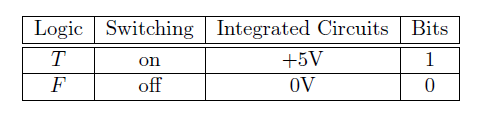
## Chapter 3

As we have seen, switches may be “on" or “off,” a terminal in an integrated circuit may read

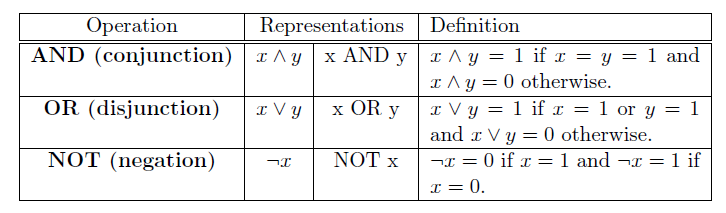
“+5 Volts" or “0 Volts," and a bit may be 1 or 0. Since switches, voltages in integrated circuits, and bits each represent two distinct states (on vs. off, +5V vs. 0V, and 1 vs. 0), each may represent the other.

Two states correspond to truth values which may be **true or false**, denoted by **T and F**, respectively.

The truth value “true" typically corresponds to “on," “+5V," and “1," while the truth value “false" typically corresponds to “off," “0V”.



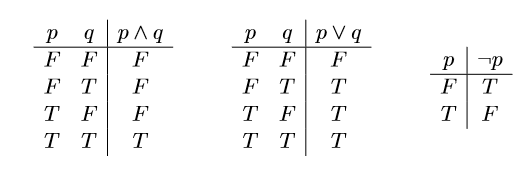
The basic operators of Boolean algebra correspond to the Basic logic gates (AND, OR, NOT).



## Truth Table

In logic, variables may take on one of ***two truth values (T or F)***, and these variables are manipulated and combined by various logical operators. The actions of these logical operators are typically defined using ***truth tables***, in a manner identical to the use of these truth tables in defining the actions of logic gates.

### The basic circuits **AND, OR, and NOT** correspond to the basic logical operators **conjunction, disjunction, and negation**, typically represented by the symbols ∧, ∨, and ¬, respectively.



**AND Rules**

|  |  |  |  |
| --- | --- | --- | --- |
| Input 1 | Input 2 | AND | Output |
| P | Q | P ∧ Q | Output |
| 0 | 0 | 0 ∧ 0 | 0 |
| 0 | 1 | 0 ∧ 1 | 0 |
| 1 | 0 | 1 ∧ 0 | 0 |
| 1 | 1 | 1 ∧ 1 | 1 |

**OR Rules**

|  |  |  |  |
| --- | --- | --- | --- |
| Input 1 | Input 2 | OR | Output |
| P | Q | P V Q | Output |
| 0 | 0 | 0 V 0 | 0 |
| 0 | 1 | 0 V 1 | 1 |
| 1 | 0 | 1 V 0 | 1 |
| 1 | 1 | 1 V 1 | 1 |

**NOT Rules**

|  |  |  |
| --- | --- | --- |
| Input | NOT | Output |
| P | ¬ P | Output |
| 0 | ¬ 0 | 1 |
| 1 | ¬ 1 | 0 |

**NAND (NOT AND) Rules: Opposite of AND operator.**

|  |  |  |
| --- | --- | --- |
| Input 1 | Input 2 | Output |
| P | Q | P NAND Q |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**NOR (NOT OR) Rules: Opposite of OR operator.**

|  |  |  |
| --- | --- | --- |
| Input 1 | Input 2 | NOR |
| P | Q | Output |
| 0 | **0** | **1** |
| 0 | **1** | **0** |
| 1 | **0** | **0** |
| 1 | **1** | **0** |

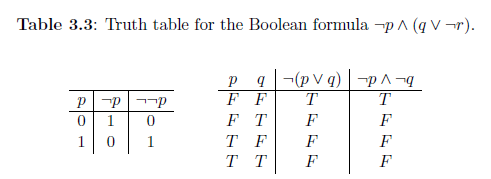
Other logical operators exist, including exclusive-OR (represented by the ⊕ symbol) and equivalence (represented by the ≡ symbol) which correspond to the XOR and XNOR.

* Truth tables can be used to represent much more than just the actions of primitive logical operators.
* Truth tables can represent the arbitrary input/output behavior of Boolean (logical) formulae or circuits. One goal of logic design is to ﬁnd eﬃcient implementations of truth tables and their corresponding Boolean formulae using logic gates.

**XOR:** Exclusive or or exclusive disjunction is a logical operation that outputs true only when inputs differ (one is true, the other is false).

### Logical Equivalent

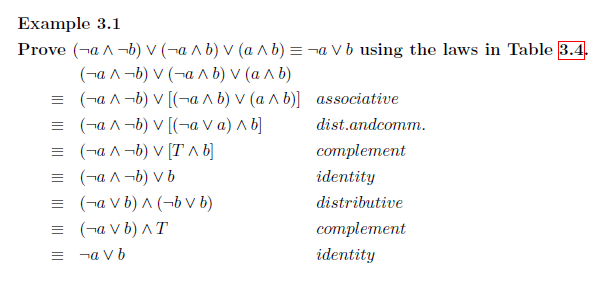
**Two Boolean formulae are said to be logically equivalent if they have the same truth table**; **they perform the same actions on their inputs**. For example, ⌐(⌐p) is logically equivalent to p and this is typically written ⌐(⌐p) ≡ p.



Two Boolean formulae can be proven logically equivalent by constructing their truth tables or through the repeated application of various laws of logic.

The laws of Boolean algebra (Table 3.4) we give here are identities that say two wff's (well-formed formula) are equivalent, such as (p^q)^r p^(q ^r). We'll see in Chapter 7 that with somewhat different notation, the same laws apply to sets.

Table 3.4: Laws of Boolean Algebra for WFFs.
Mathematical laws.



## Normal Form

#### Conjunctive Normal Form

Boolean formula is in **conjunctive normal form (CNF)** if it is **a conjunction (^) of clauses**, **where each clause is a disjunction (V) of variables.**

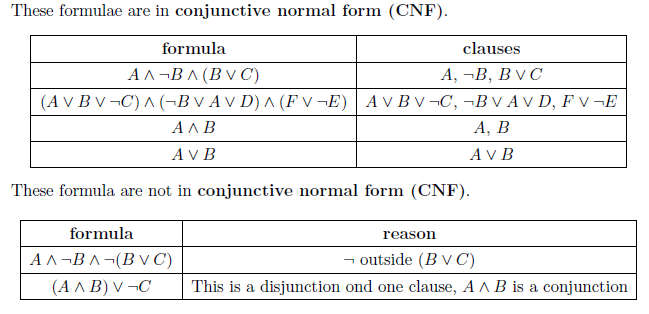
**Conjunctive normal form (CNF)** is an approach to Boolean logic that expresses formulas as conjunctions of clauses with an **AND or OR**. **Each clause connected by a conjunction, or AND, must be either a literal or contain a disjunction, or OR operator.** CNF is useful for automated theorem proving.

For example:

(A OR B) AND (C OR D)

(A OR B) AND (NOT C OR B)

Every Boolean formula can be converted into an equivalent formula that is in CNF by using the rules about logical equivalences above.



Double Negation:

1. P↔¬(¬P)P↔¬(¬P)

De Morgan's Laws

1. ¬(P⋁Q)↔(¬P)⋀(¬Q)¬(P⋁Q)↔(¬P)⋀(¬Q)
2. ¬(P⋀Q)↔(¬P)⋁(¬Q)¬(P⋀Q)↔(¬P)⋁(¬Q)

Distributive Laws

1. (P⋁(Q⋀R))↔(P⋁Q)⋀(P⋁R)(P⋁(Q⋀R))↔(P⋁Q)⋀(P⋁R)
2. (P⋀(Q⋁R))↔(P⋀Q)⋁(P⋀R)(P⋀(Q⋁R))↔(P⋀Q)⋁(P⋀R)

So let’s expand the following: (equivalent to the expression in question)

1. (((A⋀B)⋁(C⋀D))⋁E)(((A⋀B)⋁(C⋀D))⋁E) Now using 4.

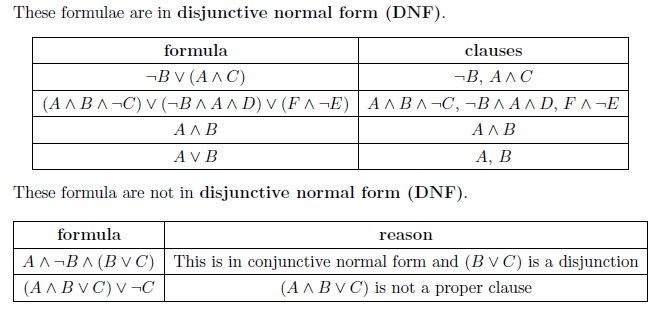
We get:

1. ((A⋀B)⋁C)⋀((A⋀B)⋁D))⋁E((A⋀B)⋁C)⋀((A⋀B)⋁D))⋁E And using 4. again
2. ((((A⋁C)⋀(B⋁C))⋀((A⋁D)⋀B⋁D)))⋁E)((((A⋁C)⋀(B⋁C))⋀((A⋁D)⋀B⋁D)))⋁E) which gives:
3. (((A⋁C)⋀(B⋁C))⋁E)⋀((A⋁D)⋀B⋁D))⋁E)(((A⋁C)⋀(B⋁C))⋁E)⋀((A⋁D)⋀B⋁D))⋁E)
4. (A⋁C⋁E)⋀(B⋁C⋁E)⋀(A⋁D⋁E)⋀(B⋁D⋁E)

#### Disjunctive Normal Form

A Boolean formula is in **disjunctive normal form (DNF)** if it is **a disjunction (V) of clauses, where each clause is a conjunction (^) of variables.**

The **disjunctive normal form** is dual to the conjunctive normal form (**ORs of ANDs** instead of ANDs of ORs) and every Boolean formula can be converted into an equivalent formula that is in DNF by using the rules about logical equivalences above.



If an expression is equivalent to a given logical expression and consists of a sum of products, then the expression is called a Disjunctive Normal Form (DNF).

A Boolean expression is in **disjunctive normal form (DNF)** if:

1. **The variables within each term are ANDed together,**
2. **The terms are ORed together, and**
3. **Every variable or its complement is represented in every term, (i.e. either A or ⌐A is in each term, B or ⌐B is in each term, etc.).**
4. **No parentheses or other Boolean operations appear in the expression.**

For example, ⌐A⌐BC + ⌐AB⌐C + ABC. (This is "not A and not B and C, or not A and B and not C, or A and B and C." This is in DNF because:

1. There are three terms; the first term is ⌐A⌐BC, the second term is ⌐AB⌐C and the third term is ABC.
2. Note that the "terms" are defined by the way the variables are ANDed and ORed.
3. Every term has either A or ⌐A, B or ⌐B, and C or ⌐C. These are the only variables in the expression.
4. There are no parentheses and no other Boolean operations (like exclusive-ors, etc.)

The following are NOT in DNF:

1. (⌐A+B+C)(A+B+⌐C)

Reason: Terms are ANDed, variables are ORed. (This is actually in conjunctive normal form)

1. ⌐AB + AB⌐C

Reason: Not every term has all variables. (First term is missing a C or ⌐C.)

1. A(BC + ⌐B⌐C)

Reason: Parentheses present.

1. ABC (+) ⌐A⌐B⌐C

Reason: Other Boolean operators present. [The '(+)' represents 'exclusive-or'.

**The rules for conjunctive normal form (CNF) are similar to the rules for DNF except for the precedence of the ANDs and ORs. A Boolean expression is in CNF if:**

1. **the variables within each term are ORed together,**
2. **the terms are ANDed together, and**
3. **Every variable or its complement is represented in every term (i.e. either A or ~A is in each term, B or ⌐B is in each term, etc.).**
4. **No parenthesis - other than those separating the terms – or other Boolean operations appear in the expression.**

For example, (⌐A+B+C)(⌐A+⌐B+⌐C)(A+⌐B+C). (This is "not A or B or C, and not A or not B or not C, and A or not B or C.") This is in CNF because:

1. 1, 2. There are three terms; the first term is⌐A+B+C, the second term is ⌐A+⌐B+⌐C and the third term is A+⌐B+C.
2. Note that the "terms" are again defined by the way the variables are ANDed and ORed, but this time the variables must be ORed within the terms, and the terms must be ANDed.
3. Every term has either A or ⌐A, B or⌐B, and C or ⌐C. These are the only variables in the expression.
4. The only parentheses are those separating the terms, and no other Boolean operations (like exclusive-ors, etc.) appear in the expression.

The following are NOT in CNF:

1. ⌐AB⌐C + ABC

Reason: Terms are ORed, variables are ANDed. (This is in DNF.)

1. ⌐A(⌐B+C)(B+⌐C)

Reason: Not every term has all variables. (First term, ~A, is missing a B or ⌐B and C or⌐C. Second and third terms each missing A or ⌐A.)

1. ⌐C(⌐A(⌐B+AC)+B)

Reason: Improper use of parentheses.

1. (A(+)B(+)C)(⌐A(+)⌐B(+)⌐C)

Reason: Other Boolean operators present. [The '(+)' represents 'exclusive-or'.]