

A [finite state machine](#) can be used as a representation of a Markov chain. Assuming a sequence of [independent and identically distributed](#) input signals (for example, symbols from a binary alphabet chosen by coin tosses), if the machine is in state y at time n , then the probability that it moves to state x at time $n + 1$ depends only on the current state.

A **finite-state machine (FSM)** or **finite-state automaton (FSA**, plural: *automata*), or simply a **state machine**, is a mathematical [model of computation](#) used to design both [computer programs](#) and [sequential logic](#) circuits. It is conceived as an [abstract machine](#) that can be in one of a finite number of [states](#). The machine is in only one state at a time; the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering event or condition; this is called a *transition*. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are [vending machines](#), which dispense products when the proper combination of coins is deposited, [elevators](#), which drop riders off at upper floors before going down, [traffic lights](#), which change sequence when cars are waiting, and [combination locks](#), which require the input of combination numbers in the proper order.

Finite-state machines can model a large number of problems, among which are [electronic design automation](#), [communication protocol](#) design, language [parsing](#) and other [engineering](#) applications. In [biology](#) and [artificial intelligence](#) research, state machines or hierarchies of state machines have been used to describe [neurological systems](#). In [linguistics](#), they are used to describe simple parts of the [grammars](#) of natural [languages](#).

Considered as an abstract model of computation, the finite state machine has less computational power than some other models of computation such as the [Turing machine](#).^[1] That is, there are tasks that no FSM can do, but some Turing machines can. This is because the FSM [memory](#) is limited by the number of states.

Concepts and terminology

A *state* is a description of the status of a system that is waiting to execute a *transition*. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. For example, when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "next" stimulus results in moving to the next station. When the system is in the "CD" state, the "next" stimulus results in moving to the next track. Identical stimuli trigger different actions depending on the current state.

In some finite-state machine representations, it is also possible to associate actions with a state:

- Entry action: performed *when entering* the state,
- Exit action: performed *when exiting* the state.

Classification

The state machines can be subdivided into Transducers, Acceptors, Classifiers and Sequencers.^[4]

Acceptors and recognizers

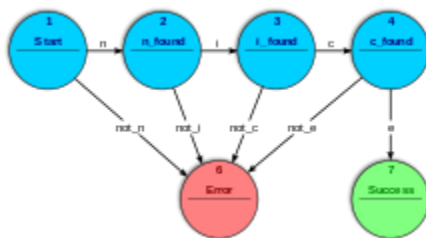


Fig. 4 Acceptor FSM: parsing the string "nice"

Acceptors (also called **recognizers** and **sequence detectors**) produce binary output, indicating whether or not received input is accepted. Each state of an FSM is either "accepting" or "not accepting". Once all input has been received, if the current state is an accepting state, the input is accepted; otherwise it is rejected. As a rule, input is a series of symbols (characters); actions are not used. The example in figure 4 shows a finite state machine that accepts the string "nice". In this FSM, the only accepting state is state 7.

A machine could also be described as defining a language, that would contain every string accepted by the machine but none of the rejected ones; that language is "accepted" by the machine. By definition, the languages accepted by FSMs are the [regular languages](#)—; a language is regular if there is some FSM that accepts it.

The problem of determining the language accepted by a given FSA is an instance of the [algebraic path problem](#)—itself a generalization of the [shortest path problem](#) to graphs with edges weighted by the elements of an (arbitrary) [semiring](#).^{[5][6][7]}

Start state

The start state is usually shown drawn with an arrow "pointing at it from any where" (Sipser (2006) p. 34).

Accept (or final) states

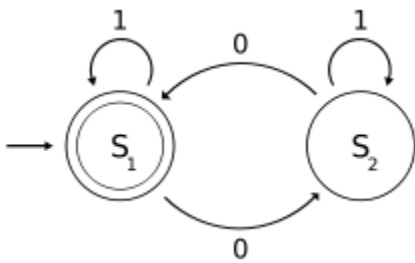


Fig. 5: Representation of a finite-state machine; this example shows one that determines whether a

binary number has an even number of 0s, where S_1 is an **accepting state**.

Accept states (also referred to as **accepting** or **final** states) are those at which the machine reports that the input string, as processed so far, is a member of the language it accepts. Accepting states are usually represented by double circles.

The start state can also be an accepting state, in which case the automaton accepts the empty string. If the start state is not an accepting state and there are no connecting edges to any of the accepting states, then the automaton is accepting nothing.

An example of an accepting state appears in Fig.5: a [deterministic finite automaton](#) (DFA) that detects whether the [binary](#) input string contains an even number of 0s.

S_1 (which is also the start state) indicates the state at which an even number of 0s has been input. S_1 is therefore an accepting state. This machine will finish in an accept state, if the binary string contains an even number of 0s (including any binary string containing no 0s). Examples of strings accepted by this DFA are ϵ (the [empty string](#)), 1, 11, 11..., 00, 010, 1010, 10110, etc...

Classifier is a generalization that, similar to acceptor, produces single output when terminates but has more than two terminal states.

Transducers

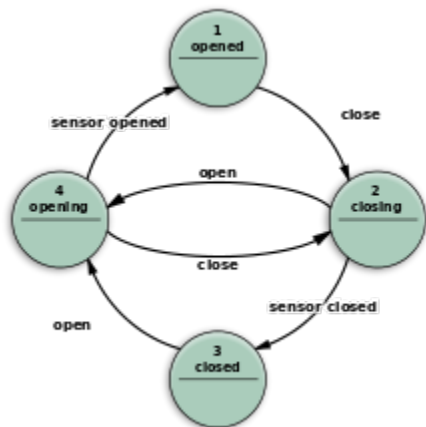


Fig. 6 Transducer FSM: Moore model example

Main article: [Finite-state transducer](#)

[Transducers](#) generate output based on a given input and/or a state using actions. They are used for control applications and in the field of [computational linguistics](#).

In control applications, two types are distinguished:

[Moore machine](#)

The FSM uses only entry actions, i.e., output depends only on the state. The advantage of the Moore model is a simplification of the behaviour. Consider an elevator door. The state machine recognizes two commands: "command_open" and "command_close", which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" stop the motor when fully opened or closed. They signal to the outside world (e.g., to other state machines) the situation: "door is open" or "door is closed".

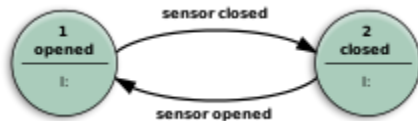


Fig. 7 Transducer FSM: Mealy model example

[Mealy machine](#)

The FSM uses only input actions, i.e., output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 7 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work, e.g., for [virtual FSM](#) but not for [event-driven FSM](#)). There are two input actions (I:): "start motor to close the door if command_close arrives" and "start motor in the other direction to open the door if command_open arrives". The "opening" and "closing" intermediate states are not shown.

Generators

The **sequencers** or **generators** are a subclass of aforementioned types that have a [single-letter input alphabet](#). They produce only one sequence, which can be interpreted as output sequence of transducer or classifier outputs.

Determinism

A further distinction is between **deterministic** ([DFA](#)) and **non-deterministic** ([NFA](#), [GNFA](#)) automata. In deterministic automata, every state has exactly one transition for each possible

input. In non-deterministic automata, an input can lead to one, more than one or no transition for a given state. This distinction is relevant in practice, but not in theory, as there exists an algorithm (the [powerset construction](#)) that can transform any NFA into a more complex DFA with identical functionality.

The FSM with only one state is called a combinatorial FSM and uses only input actions. This concept is useful in cases where a number of FSM are required to work together, and where it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools.^[8]

Alternative semantics

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers.^[9] They combine [hierarchical state machines](#), flow graphs, and [truth tables](#) into one language, resulting in a different formalism and set of semantics.^[10] Figure 8 illustrates this mix of state machines and flow graphs with a set of states to represent the state of a stopwatch and a flow graph to control the ticks of the watch. These charts, like Harel's original state machines,^[11] support hierarchically nested states, [orthogonal regions](#), state actions, and transition actions.^[12]

In [computer science](#), an **abstract state machine (ASM)** is a [state machine](#) operating on [states](#) which are arbitrary data structures ([structure](#) in the sense of [mathematical logic](#), that is a nonempty [set](#) together with a number of [functions](#) ([operations](#) over the set) and [relations](#)).

The **ASM Method** is a practical and scientifically well-founded [systems engineering](#) method which bridges the gap between the two ends of system development:

- the human understanding and formulation of real-world problems ([requirements capture](#) by accurate high-level modeling at the level of abstraction determined by the given application domain)
- the deployment of their algorithmic solutions by code-executing machines on changing platforms (definition of design decisions, system and implementation details).

The method builds upon three basic concepts:

- *ASM*: a precise form of pseudo-code, generalizing [Finite State Machines](#) to operate over arbitrary data structures
- *ground model*: a rigorous form of blueprints, serving as authoritative reference model for the design
- *refinement*: a most general scheme for stepwise instantiations of model abstractions to concrete system elements, providing controllable links between the more and more detailed descriptions at the successive stages of system development.

In the original conception of ASMs, a single [agent](#) executes a program in a sequence of steps, possibly interacting with its environment. This notion was extended to capture [distributed computations](#), in which multiple agents execute their programs concurrently.

Since ASMs model algorithms at arbitrary levels of abstraction, they can provide high-level, low-level and mid-level views of a hardware or software design. ASM specifications often consist of a series of ASM models, starting with an abstract *ground model* and proceeding to greater levels of detail in successive [refinements](#) or coarsenings.

Due to the algorithmic and mathematical nature of these three concepts, ASM models and their properties of interest can be analyzed using any rigorous form of [verification](#) (by reasoning) or [validation](#) (by experimentation, testing model executions).

In [computer science](#), a **communicating finite-state machine** is a [finite state machine](#) labeled with "receive" and "send" operations over some alphabet of channels. They were introduced by Brand and Zafiropulo,^[1] and can be used as a model of [concurrent](#) processes like [Petri nets](#). Communicating finite state machines are used frequently for modeling a communication protocol since they make it possible to detect major protocol design errors, including boundedness, deadlocks, and unspecified receptions.^[2]

The advantage of communicating finite state machines is that they make it possible to decide many properties in communication protocols, beyond the level of just detecting such properties. This advantage rules out the need for human assistance or restriction in generality.^[3]

It has been proved with the introduction of the concept itself that when two finite state machines communicate with only one type of messages, boundedness, deadlocks, and unspecified reception state can be decided and identified while such is not the case when the machines communicate with two or more types of messages. Later, it has been further proved that when only one finite state machine communicates with single type of message while the communication of its partner is unconstrained, we can still decide and identify boundedness, deadlocks, and unspecified reception state.^[4]

It has been further proved that when the message priority relation is empty, boundedness, deadlocks and unspecified reception state can be decided even under the condition in which there are two or more types of messages in the communication between finite state machines.^[5]

Boundedness, deadlocks, and unspecified reception state are all decidable in polynomial time (which means that a particular problem can be solved in tractable, not infinite, amount of time) since the decision problems regarding them are nondeterministic logspace complete.^[6]

Communicating finite state machines can be the most powerful in situations where the propagation delay is not negligible (so that several messages can be in transit at one time) and in situations where it is natural to describe the protocol parties and the communication medium as separate entities.^[7]

Communicating Hierarchical State Machine

Hierarchical state machines are finite state machines whose states themselves can be other machines. Since a communicating finite state machine is characterized by concurrency, the most notable trait in a **communicating hierarchical state machine** is the coexistence of hierarchy and concurrency. This had been considered highly suitable as it signifies stronger interaction inside the machine.

However, it was proved that the coexistence of hierarchy and concurrency intrinsically costs language inclusion, language equivalence, and all of universality

In a conventional [finite state machine](#), the transition is associated with a set of input [Boolean](#) conditions and a set of output Boolean functions. In an **extended finite state machine (EFSM) model**, the transition can be expressed by an “[if statement](#)” consisting of a set of [trigger conditions](#). If trigger conditions are all satisfied, the transition is fired, bringing the machine from the current state to the next state and performing the specified [data operations](#).

Structure

EFSM Architecture: An EFSM model consists of the following three major combinational blocks (and a few registers).

- FSM-block: A conventional finite state machine realizing the state transition graphs of the EFSM model.
- A-block: an arithmetic block for performing the data operation associated with each transition. The operation of this block is regulated by the output signals of the FSM block.
- E-block: A block for evaluating the trigger conditions associated with each transition. The input signals to this block are the data variables, while the output is a set of [binary](#) signals taken for input by the FSM-block. Information about redundant computation is extracted by analyzing the interactions among the three basic blocks. Using this information, certain input operands of the [arithmetic](#) block and [evaluation](#) block can be frozen through input gating under specific run time conditions to reduce the unnecessary switching in the design. At the architecture level, if each trigger evaluation & data operation is regarded as an atomic action, then the EFSM implies an almost lowest-power implementation.

The cycle behavior of an EFSM can be divided into three steps:

1. In E-block, evaluate all trigger conditions.
2. In FSM-block, compute the next state & the signals controlling A-block.
3. In A-block, perform the necessary data operations and data movements.

Definition [\[edit\]](#)

An EFSM is defined^[1] as a 7-tuple $M = (I, O, S, D, F, U, T)$ where

- S is a set of symbolic states,
- I is a set of input symbols,
- O is a set of output symbols,
- D is an n -dimensional linear space $D_1 \times \dots \times D_n$,
- F is a set of enabling functions $f_i : D \rightarrow \{0, 1\}$,
- U is a set of update functions $u_i : D \rightarrow D$,
- T is a transition relation, $T : S \times F \times I \rightarrow S \times U \times O$

A **Finite State Machine with Datapath** (FSMD) is a [mathematical abstraction](#) that is sometimes used to design [digital logic](#) or [computer programs](#).

An FSMD is a digital system composed of a [finite-state machine](#), which controls the [program flow](#), and a [datapath](#), which performs data processing operations.

FSMDs are essentially sequential programs in which statements have been scheduled into states, thus resulting in more complex state diagrams.

Here, a program is converted into a complex state diagram in which states and arcs may include [arithmetic expressions](#), and those expressions may use external inputs and outputs as well as variables.

FSMs do not use variables or arithmetic operations/conditions, thus FSMDs are more powerful than FSMs.

The FSMD level of abstraction is often referred to as the [register-transfer level](#).

FSMD is equivalent to [Turing machine](#) in power.

In [electronics](#), a **logic gate** is an idealized or physical device implementing a [Boolean function](#); that is, it performs a [logical operation](#) on one or more logical inputs, and produces a single logical output. Depending on the context, the term may refer to an **ideal logic gate**, one that has for instance zero [rise time](#) and unlimited [fan-out](#), or it may refer to a non-ideal physical device^[1] (see [Ideal and real op-amps](#) for comparison).

Logic gates are primarily implemented using [diodes](#) or [transistors](#) acting as [electronic switches](#), but can also be constructed using [vacuum tubes](#), electromagnetic [relays](#) ([relay logic](#)), [fluidic logic](#), [pneumatic logic](#), [optics](#), [molecules](#), or even [mechanical](#) elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of [Boolean logic](#), and therefore, all of the algorithms and [mathematics](#) that can be described with Boolean logic.

Logic circuits include such devices as [multiplexers](#), [registers](#), [arithmetic logic units](#) (ALUs), and [computer memory](#), all the way up through complete [microprocessors](#), which may contain more than 100 million gates. In modern practice, most gates are made from [field-effect transistors](#) (FETs), particularly [MOSFETs](#) (metal–oxide–semiconductor field-effect transistors).

Compound logic gates [AND-OR-Invert](#) (AOI) and OR-AND-Invert (OAI) are often employed in circuit design because their construction using MOSFETs is simpler and more efficient than the sum of the individual gates.^[2]

In [reversible logic](#), [Toffoli gates](#) are used.

Electronic gates

Main article: [Logic family](#)

To build a [functionally complete](#) logic system, [relays](#), [valves](#) (vacuum tubes), or [transistors](#) can be used. The simplest family of logic gates using [bipolar transistors](#) is called [resistor-transistor logic](#) (RTL). Unlike simple diode logic gates (which do not have a gain element), RTL gates can be cascaded indefinitely to produce more complex logic functions. RTL gates were used in early [integrated circuits](#). For higher speed and better density, the resistors used in RTL were replaced by diodes resulting in [diode-transistor logic](#) (DTL). [Transistor-transistor logic](#) (TTL) then supplanted DTL. As integrated circuits became more complex, bipolar transistors were replaced with smaller [field-effect transistors](#) ([MOSFETs](#)); see [PMOS](#) and [NMOS](#). To reduce power consumption still further, most contemporary chip implementations of digital systems now use [CMOS](#) logic. CMOS uses complementary (both n-channel and p-channel) MOSFET devices to achieve a high speed with low power dissipation.

For small-scale logic, designers now use prefabricated logic gates from families of devices such as the [TTL 7400 series](#) by [Texas Instruments](#), the [CMOS 4000 series](#) by [RCA](#), and their more recent descendants. Increasingly, these fixed-function logic gates are being replaced by [programmable logic devices](#), which allow designers to pack a large number of mixed logic gates into a single integrated circuit. The field-programmable nature of [programmable logic devices](#) such as [FPGAs](#) has removed the 'hard' property of hardware; it is now possible to change the logic design of a hardware system by reprogramming some of its components, thus allowing the features or function of a hardware implementation of a logic system to be changed.

Other types of logic gates include, but are not limited to ^[3]

Tunnel diode logic	TDL	Exactly the same as Diode Logic but can perform at a faster speed.	Diode logic
Neon Logic	NL	Uses neon bulbs or 3 element neon trigger tubes to perform logic.	
Core Diode Logic	CDL	Performed by semiconductor diodes and small ferrite toroidal cores for moderate speed and moderate power level.	
4Layer Device Logic	4LDL	Uses thyristors and SCR's to perform logic operations where high current and or high voltages are required.	
Direct Coupled Transistor Logic	DCTL	Uses transistors switching between bottom saturated and tunnel states to perform logic, depends upon use of transistors with carefully controlled parameters. Economical but tends to be susceptible to noise because of the lower voltage levels employed. Often considered to be the father to modern TTL logic.	Direct-coupled transistor logic
Current Mode Logic	CML	Uses transistors to perform logic but biasing is from constant current sources to prevent saturation and allow extremely fast switching. Has high noise immunity despite fairly low logic levels.	Current-mode logic
Diode logic	DL		Diode logic

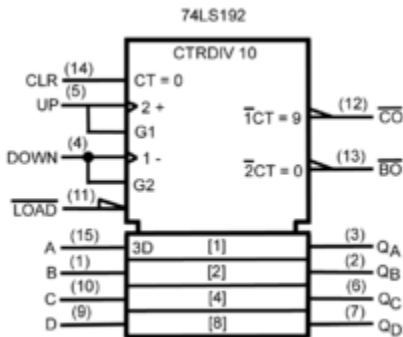
Electronic logic gates differ significantly from their relay-and-switch equivalents. They are much faster, consume much less power, and are much smaller (all by a factor of a million or more in most cases). Also, there is a fundamental structural difference. The switch circuit creates a continuous metallic path for current to flow (in either direction) between its input and its output. The semiconductor logic gate, on the other hand, acts as a high-[gain voltage amplifier](#), which sinks a tiny current at its input and produces a low-impedance voltage at its output. It is not possible for current to flow between the output and the input of a semiconductor logic gate.

Another important advantage of standardized integrated circuit logic families, such as the 7400 and 4000 families, is that they can be cascaded. This means that the output of one gate can be wired to the inputs of one or several other gates, and so on. Systems with varying degrees of complexity can be built without great concern of the designer for the internal workings of the gates, provided the limitations of each integrated circuit are considered.

The output of one gate can only drive a finite number of inputs to other gates, a number called the '[fanout](#) limit'. Also, there is always a delay, called the '[propagation delay](#)', from a change in input of a gate to the corresponding change in its output. When gates are cascaded, the total propagation delay is approximately the sum of the individual delays, an effect which can become a problem in high-speed circuits. Additional delay can be caused when a large number of inputs are connected to an output, due to the distributed [capacitance](#) of all the inputs and wiring and the finite amount of current that each output can provide.

Symbols

Symbols



A synchronous 4-bit up/down decade counter symbol (74LS192) in accordance with ANSI/IEEE Std. 91-1984 and IEC Publication 60617-12.


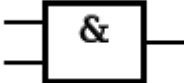

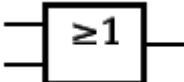

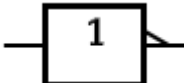
There are two sets of symbols for elementary logic gates in common use, both defined in [ANSI/IEEE](#) Std 91-1984 and its supplement ANSI/IEEE Std 91a-1991. The "distinctive shape" set, based on traditional schematics, is used for simple drawings, and derives from MIL-STD-806 of the 1950s and 1960s. It is sometimes unofficially described as "military", reflecting its origin. The "rectangular shape" set, based on ANSI Y32.14 and other early industry standards, as later refined by IEEE and IEC, has rectangular outlines for all types of gate and allows representation of a much wider range of devices than is possible with the traditional symbols.^[4] The IEC standard, IEC 60617-12, has been adopted by other standards, such as [EN](#) 60617-12:1999 in Europe, [BS](#) EN 60617-12:1999 in the United Kingdom, and DIN EN 60617-12:1998 in Germany.

The mutual goal of IEEE Std 91-1984 and IEC 60617-12 was to provide a uniform method of describing the complex logic functions of digital circuits with schematic symbols. These functions were more complex than simple AND and OR gates. They could be medium scale circuits such as a 4-bit counter to a large scale circuit such as a microprocessor.

IEC 617-12 and its successor IEC 60617-12 do not explicitly show the "distinctive shape" symbols, but do not prohibit them.^[4] These are, however, shown in ANSI/IEEE 91 (and 91a) with this note: "The distinctive-shape symbol is, according to IEC Publication 617, Part 12, not preferred, but is not considered to be in contradiction to that standard." IEC 60617-12 correspondingly contains the note (Section 2.1) "Although non-preferred, the use of other symbols recognized by official national standards, that is distinctive shapes in place of symbols [list of basic gates], shall not be considered to be in contradiction with this standard. Usage of these other symbols in combination to form complex symbols (for example, use as embedded symbols) is discouraged." This compromise was reached between the respective IEEE and IEC working groups to permit the IEEE and IEC standards to be in mutual compliance with one another.

A third style of symbols was in use in Europe and is still widely used in European academia. See the column "DIN 40700" in [the table in the German Wikipedia](#).

In the 1980s, schematics were the predominant method to design both [circuit boards](#) and custom ICs known as [gate arrays](#). Today custom ICs and the [field-programmable gate array](#) are typically designed with [Hardware Description Languages](#) (HDL) such as [Verilog](#) or [VHDL](#).

Type	Distinctive shape (IEEE Std 91/91a-1991)	Rectangular shape (IEEE Std 91/91a-1991 IEC 60617-12 : 1997)	Boolean algebra between A & B	Truth table																		
AND				<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A AND B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	INPUT		OUTPUT	A	B	A AND B	0	0	0	0	1	0	1	0	0	1	1	1
INPUT		OUTPUT																				
A	B	A AND B																				
0	0	0																				
0	1	0																				
1	0	0																				
1	1	1																				
OR				<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A OR B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	INPUT		OUTPUT	A	B	A OR B	0	0	0	0	1	1	1	0	1	1	1	1
INPUT		OUTPUT																				
A	B	A OR B																				
0	0	0																				
0	1	1																				
1	0	1																				
1	1	1																				
NOT				<table><tr><th>INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>NOT A</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	INPUT	OUTPUT	A	NOT A	0	1	1	0										
INPUT	OUTPUT																					
A	NOT A																					
0	1																					
1	0																					

In electronics a NOT gate is more commonly called an inverter. The circle on the symbol is called a

bubble, and is used in logic diagrams to indicate a logic negation between the external logic state and the internal logic state (1 to 0 or vice versa). On a circuit diagram it must be accompanied by a statement asserting that the *positive logic convention* or *negative logic convention* is being used (high voltage level = 1 or high voltage level = 0, respectively). The *wedge* is used in circuit diagrams to directly indicate an active-low (high voltage level = 0) input or output without requiring a uniform convention throughout the circuit diagram. This is called *Direct Polarity Indication*. See IEEE Std 91/91A and IEC 60617-12. Both the *bubble* and the *wedge* can be used on distinctive-shape and [rectangular](#)-shape symbols on circuit diagrams, depending on the logic convention used. On pure logic diagrams, only the *bubble* is meaningful.

NAND



INPUT		OUTPUT
A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

NOR



INPUT		OUTPUT
A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

XOR



INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1

1	0	1
1	1	0

The output of a two input exclusive-OR is true only when the two input values are *different*, and false if they are equal, regardless of the value. If there are more than two inputs, the output of the distinctive-shape symbol is undefined. The output of the rectangular-shaped symbol is true if the number of true inputs is exactly one or exactly the number following the "=" in the qualifying symbol. In practice, these gates are built from combinations of simpler logic gates.

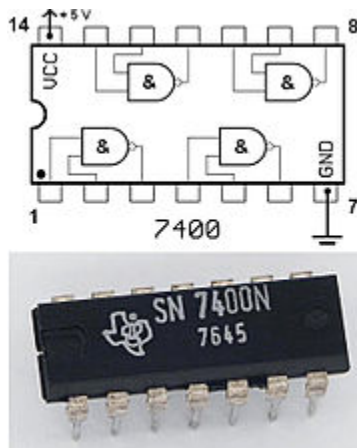
XNOR



INPUT		OUTPUT
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

Universal logic gates

For more details on the theoretical basis, see [Functional completeness](#).



The 7400 chip, containing four NANDs. The two additional pins supply power (+5 V) and connect the ground.

[Charles Sanders Peirce](#) (during 1880–81) showed that [NOR gates](#) alone (or alternatively [NAND gates](#) alone) can be used to reproduce the functions of all the other logic gates, but his work on it was unpublished until 1933.^[5] The first published proof was by [Henry M. Sheffer](#) in 1913, so the NAND logical operation is sometimes called [Sheffer stroke](#); the [logical NOR](#) is sometimes called *Peirce's arrow*.^[6] Consequently, these gates are sometimes called *universal logic gates*.^[7]

De Morgan equivalent symbols

By use of [De Morgan's laws](#), an *AND* function is identical to an *OR* function with negated inputs and outputs. Likewise, an *OR* function is identical to an *AND* function with negated inputs and outputs. A NAND gate is equivalent to an OR gate with negated inputs, and a NOR gate is equivalent to an AND gate with negated inputs.

This leads to an alternative set of symbols for basic gates that use the opposite core symbol (*AND* or *OR*) but with the inputs and outputs negated. Use of these alternative symbols can make logic circuit diagrams much clearer and help to show accidental connection of an active high output to an active low input or vice versa. Any connection that has logic negations at both ends can be replaced by a negationless connection and a suitable change of gate or vice versa. Any connection that has a negation at one end and no negation at the other can be made easier to interpret by instead using the De Morgan equivalent symbol at either of the two ends. When negation or polarity indicators on both ends of a connection match, there is no logic negation in that path (effectively, bubbles "cancel"), making it easier to follow logic states from one symbol to the next. This is commonly seen in real logic diagrams - thus the reader must not get into the habit of associating the shapes exclusively as OR or AND shapes, but also take into account the bubbles at both inputs and outputs in order to determine the "true" logic function indicated.

A De Morgan symbol can show more clearly a gate's primary logical purpose and the polarity of its nodes that are considered in the "signaled" (active, on) state. Consider the simplified case where a two-input NAND gate is used to drive a motor when either of its inputs are brought low by a switch. The "signaled" state (motor on) occurs when either one OR the other switch is on. Unlike a regular NAND symbol, which suggests AND logic, the De Morgan version, a two negative-input OR gate, correctly shows that OR is of interest. The regular NAND symbol has a bubble at the output and none at the inputs (the opposite of the states that will turn the motor on), but the De Morgan symbol shows both inputs and output in the polarity that will drive the motor.

De Morgan's theorem is most commonly used to implement logic gates as combinations of only NAND gates, or as combinations of only NOR gates, for economic reasons.

Data storage

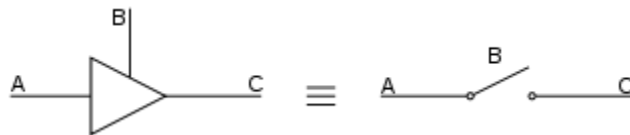
Main article: [Sequential logic](#)

Logic gates can also be used to store data. A storage element can be constructed by connecting several gates in a "[latch](#)" circuit. More complicated designs that use [clock signals](#) and that change only on a rising or falling edge of the clock are called edge-triggered "[flip-flops](#)".

Formally, a flip-flop is called a bistable circuit, because it has two stable states which it can maintain indefinitely. The combination of multiple flip-flops in parallel, to store a multiple-bit value, is known as a register. When using any of these gate setups the overall system has memory; it is then called a [sequential logic](#) system since its output can be influenced by its previous state(s), i.e. by the *sequence* of input states. In contrast, the output from [combinatorial logic](#) is purely a combination of its present inputs, unaffected by the previous input and output states.

These logic circuits are known as computer [memory](#). They vary in performance, based on factors of [speed](#), complexity, and reliability of storage, and many different types of designs are used based on the application.

Three-state logic gates



A tristate buffer can be thought of as a switch. If *B* is on, the switch is closed. If *B* is off, the switch is open.

Main article: [Tri-state buffer](#)

A three-state logic gate is a type of logic gate that can have three different outputs: high (H), low (L) and high-impedance (Z). The high-impedance state plays no role in the logic, which is strictly binary. These devices are used on [buses](#) of the [CPU](#) to allow multiple chips to send data. A group of three-states driving a line with a suitable control circuit is basically equivalent to a [multiplexer](#), which may be physically distributed over separate devices or plug-in cards.

In electronics, a high output would mean the output is sourcing current from the positive power terminal (positive voltage). A low output would mean the output is sinking current to the negative power terminal (zero voltage). High impedance would mean that the output is effectively disconnected from the circuit.