

Electronics

A Guide for Synthetic Biologists

Peking University, China

2017 iGEMer

Preface

Since the beginning of synthetic biology, it has been tightly bound with electronics. The design and implementation of the genetic repressilator and toggle switch gives us hope that one day we could engineer microbes to perform functions of electronic devices, even computers. Though we still have a long way to go, advances in synthetic biology have made us enthusiastic about the future of realizing the criteria of complete rational design in biotechnology.

Every living cell within us is a hybrid analog-digital supercomputer that implements highly computationally intensive nonlinear, stochastic, differential equations with 30,000 gene-protein state variables that interact via complex feedback loops. Even at the end of Moore's law, we will not match such performance by even a few orders of magnitude. To understand the basic mechanism of cells require the knowledge of biotechnology. However, we cannot deny the possibility that such computing power shares basic principles with our mathematical theories or electronic design.

The field of synthetic biology is much more ambitious. It attempts to transfer engineering design principles and experimental techniques into rational biological design. This represents the ultimate limit of Moore's law: computation with the molecules themselves at the nanoscale by controlled biochemistry and biophysics. To achieve this ultimate goal, it is necessary to take and modify mature theories and their products already there in the electronics field.

A great number of synthetic biologists have built and characterized a variety of logic gates in a number of organisms, and are familiar with combinational logic, Boolean algebra and design automation, which used to be terms of electrical engineering and computer science. On the other side, students and researchers focusing on electronic engineering and computer science gained a much deeper understanding of the models from biological research, such as neural networks. However, we found that such

interchange between two subjects usually stops at some specific concepts, and lacks a general view of the development and the frontiers of the subjects.

Therefore, communication channels should be established and mutual understanding should be promoted. Through the interviews with two experts, we recognized that both sides showed a strong willingness to know the other domain more. This, however, was confronted with some barriers, from basic concepts to perspectives of analysis. We, as a team of mixed backgrounds, pioneered the bridging between two types of researchers. Then comes a handbook for synthetic biologists, which clarifies basic units as well as design principles in electrical engineering. We hope it will help inspire novel ideas, designs, and analytical views.

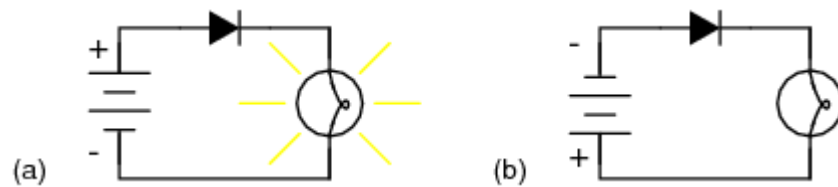
CONTENT

Analogous Circuit	1
Diode	1
Amplifier	2
Comparator	3
Digital Circuit	4
Combinational Logic Functions	4
Boolean Algebra	4
Truth table	5
Logic Gates	6
Switches	11
Multiplexers and Demultiplexes	12
Encoders and decoders	16
Sequential Circuits	19
State	19
Clock	20
Flip-Flop	21
Asynchronous Counters	24

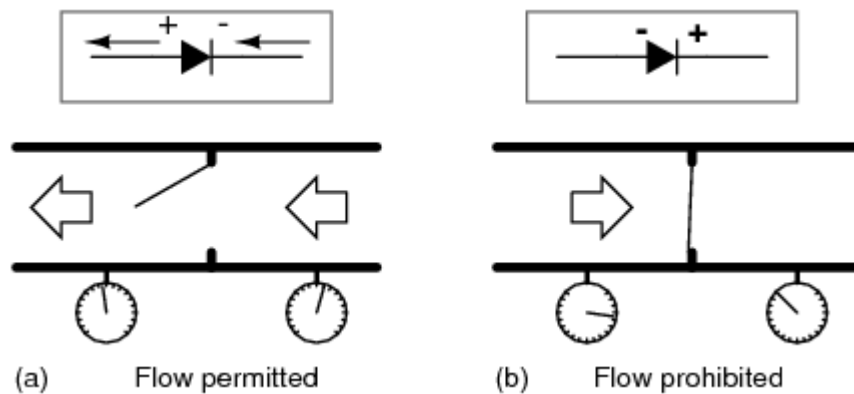
Analogous Circuit

Diode

In electronics, a diode is a two-terminal electronic component that conducts primarily in one direction (asymmetric conductance); it has low (ideally zero) resistance to the current in one direction, and high (ideally infinite) resistance in the other. In other words, a diode allows an electric current to pass in forward direction (forward-biased), while blocking current in the reverse direction (reverse-biased). Thus, the diode can be viewed as an electronic version of a check valve.

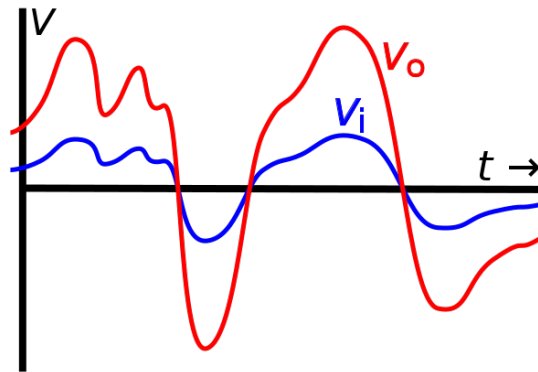


A diode may be thought of as like a switch: “closed” when forward-biased and “open” when reverse-biased.



Amplifier

An amplifier, electronic amplifier is an electronic device that can increase the power of a signal (a time-varying voltage or current). An amplifier uses electric power from a power supply to increase the amplitude of a signal. The amount of amplification provided by an amplifier is measured by its gain: the ratio of output to input. An amplifier is a circuit that can give a power gain greater than one.



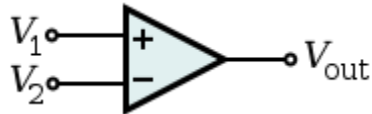
V_i -input

V_o -output

Comparator

In electronics, a comparator is a device that compares two voltages or currents and outputs a digital signal indicating which is larger. It has two analog input terminals V_+ and V_- , and one binary digital output V_o . The output is ideally

$$V_o = \begin{cases} 1, & \text{if } V_+ > V_- \\ 0, & \text{if } V_+ < V_- \end{cases}$$



Digital Circuit

Combinational Logic Functions

Boolean Algebra

Boolean algebra is the branch of algebra in which the values of the variables are the truth values true and false, usually denoted 1 and 0 respectively. It is thus a formalism for describing logical relations in the same way that ordinary algebra describes numeric relations.

A Boolean function is $B_k \rightarrow B_m$, which is the set of all functions that map k bit inputs to m bit outputs, where $k \geq 0$ and $m > 0$.

All arithmetic operations performed with Boolean quantities have but one of two possible outcomes: either 1 or 0. Instead of elementary algebra where the values of the variables are numbers, and the prime operations are addition and multiplication, the main operations of Boolean algebra are the conjunction and denoted as \wedge , the disjunction or denoted as \vee , and the negation not denoted as \neg . Consequently, the "Laws" of Boolean algebra often differ from the "Laws" of real-number algebra, making possible such statements as $1 + 1 = 1$, which would normally be considered absurd.

Boolean numbers are not the same as binary numbers. Whereas Boolean numbers represent an entirely different system of mathematics from real numbers, binary is nothing more than an alternative notation for real numbers. The difference is that Boolean quantities are restricted to a single bit (either 1 or 0), whereas binary numbers may be composed of many bits adding up in place-weighted form to a value of any finite size.

Truth table

Truth tables are one way to define a Boolean function. Here is an example of a truth table. Input variables start with the letter x. Output variables start with the letter z.

x2	x1	x0	z2	z1	z0
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	1	1	1

Since there are 3 input variables, there are $2^3 = 8$ possible 3-bit patterns. Thus, there are 8 rows. This truth table is a function of the following type $B^3 \rightarrow B^3$. That is, it's a function which is an element of the set of functions from 3-bit inputs to 3-bit outputs.

Logic Gates

Basic gates

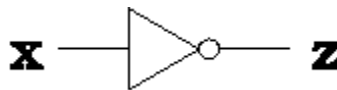
Logic gates serve as the building blocks to digital logic circuits using combinational logic. We're going to consider the following gates: NOT gates (also called inverters), AND gates, OR gates, NAND gates, NOR gates, XOR gates, and XNOR gates.

One common way to express the particular function of a gate circuit is called a truth table. Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low," "1" or "0," for each input terminal of the gate), along with the corresponding output logic level, either "high" or "low."

NOT

NOT gates or inverters have a single bit input and a single bit of output.

This is a diagram of a NOT gate. It is a triangle with a circle on the right. The circle indicates "negation".



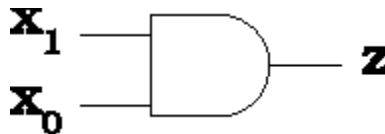
Truth table of NOT gate

X	Z
1	1
0	0

X, is the input and z is the output.

AND

The output of AND gate is 1 only if all inputs are 1. Otherwise, the output is 0.



Truth table of AND

x1	x0	z
0	0	0
0	1	0
1	0	0
1	1	1

The function's properties:

- □ The function is symmetric. $x * y == y * x$. ("*" represent AND)
- □ The function is associative. $(x * y) * z == x * (y * z)$.

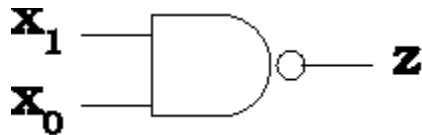
Definition of an n-input AND gate.

$$\text{AND}(x_1, x_2, \dots, x_n) = x_1 * x_2 * \dots * x_n$$

NAND

The definition of NAND is based on AND. NAND is the negation of AND.

$$\text{NAND}(x_1, x_2, \dots, x_n) = \text{NOT}(\text{AND}(x_1, x_2, \dots, x_n))$$



Truth table of NAND

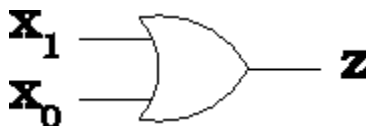
x1	x0	z
0	0	1
0	1	1
1	0	1
1	1	0

The function's properties:

- □ The function is symmetric. $x \text{ NAND } y == y \text{ NAND } x$.
- □ The function is not associative.

OR

The output of OR gate is 0 only if all inputs are 0. Otherwise, the output is 1.



Truth table of OR gate

x1	x0	z
0	0	0
0	1	1

1	0	1
1	1	1

The function's properties:

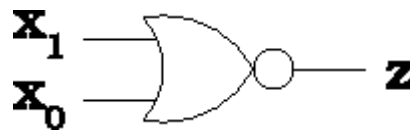
- □ The function is symmetric. $x + y == y + x$. ("+" represent OR)
- □ The function is associative. $(x + y) + z == x + (y + z)$.

Definition of an n-input OR gate.

$$\text{OR}(x_1, x_2, \dots, x_n) = x_1 + x_2 + \dots + x_n$$

NOR

The output of NOR gate is the negation of OR.



Truth table of NOR gate

x1	x0	z
0	0	1
0	1	0
1	0	0
1	1	0

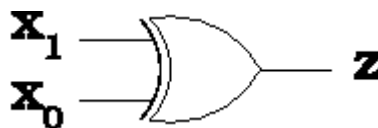
The function's properties:

- □ The function is symmetric. $x \text{ NOR } y == y \text{ NOR } x$.
- □ The function is not associative.

XOR

The output of XOR gate is 1 only if the inputs have opposite values. That is, when one input has value 0, and the other has value 1. Otherwise, the output is 0. XOR can be defined using AND, OR, and NOT.

$$x \text{ XOR } y == (x \text{ AND } (\text{NOT } y)) \text{ OR } ((\text{NOT } x) \text{ AND } y) == x \setminus y + y \setminus x$$



Truth table of XOR gate

x1	x0	z
0	0	0
0	1	1
1	0	1
1	1	0

The function's properties:

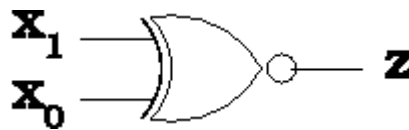
- □ The function is symmetric. $x \oplus y == y \oplus x$. ("⊕" represent XOR)
- □ The function is associative. $[x \oplus y] \oplus z == x \oplus [y \oplus z]$.

Definition of an n-input XOR gate.

$$\text{XOR}(x_1, x_2, \dots, x_n) = x_1 \oplus x_2 \oplus \dots \oplus x_n$$

XNOR

The output of XNOR gate is the negation of XOR and has 1 when all inputs are the same.



Truth table of XNOR gate

x1	x0	z
0	0	0
0	1	1
1	0	1
1	1	0

The function's properties:

- □ The function is symmetric. Thus, $x \text{ XNOR } y == y \text{ XNOR } x$.
- □ The function is associative. Thus, $(x \text{ XNOR } y) \text{ XNOR } z == x \text{ XNOR } (y \text{ XNOR } z)$.

Definition of an n-input NXOR gate.

$$\text{XNOR}(x_1, x_2, \dots, x_n) = x_1 \text{ XNOR } x_2 \text{ XNOR } \dots \text{ XNOR } x_n$$

Building Circuits with Gates







We can use logic gates to build circuits. While we've described 6 gates, you can do with only two gates to build all possible circuits. These circuits can implement any truth table.

Gate Delay

Real gates have delay. It means if you change the value of the inputs (from 0 and 0 to 0 and 1) then the output takes some small amount of time before it changes. This delay is called gate delay. This delay is due to the fact that the time it takes to do the computation is not infinitely quick. This delay limits how fast the inputs can change and yet the output has meaningful values.

Switches

An electrical switch is any device used to interrupt the flow of electrons in a circuit. Switches are essentially binary devices: they are either completely on ("closed") or completely off ("open"). There are many different types of switches.

<p>Toggle switches are actuated by a lever angled in one of two or more positions.</p>	<p style="text-align: center;">Toggle switch</p> 
<p>Pushbutton switches are two-position devices actuated with a button that is pressed and released.</p>	<p style="text-align: center;">Pushbutton switch</p> 
<p>Selector switches are actuated with a rotary knob or lever of some sort to select one of two or more positions.</p>	<p style="text-align: center;">Selector switch</p> 
<p>Gas or liquid pressure can be used to actuate a switch mechanism if that pressure is applied to a piston, diaphragm, or bellows, which converts pressure to mechanical force.</p>	<p style="text-align: center;">Pressure switch</p> 
<p>A floating object can be used to actuate a switch mechanism when the liquid level in a tank rises past a certain point.</p>	<p style="text-align: center;">Liquid level switch</p> 
<p>Inserted into a pipe, a flow switch will detect any gas or liquid flow rate</p>	<p style="text-align: center;">Liquid flow switch</p> 

Multiplexers and Demultiplexes

Multiplexers

A $n-1$ multiplexer (MUX) is a device that picks one of n inputs and direct it to an output.

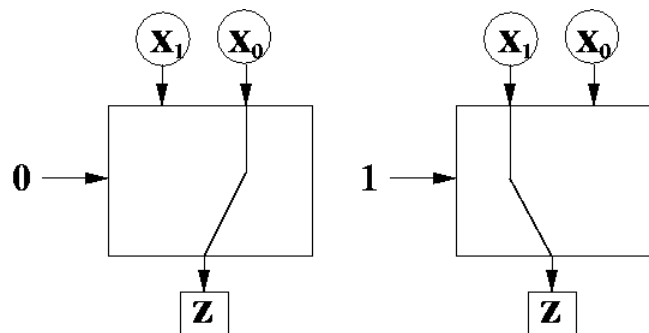
An $n-1$ MUX consists of the following:

- □ Data inputs: n
- □ Control inputs: $\text{ceil}(\log_2 n)$
- □ Outputs: 1

where ceil is the ceiling function ($\text{ceil}(x) = n$ for the smallest integer where $n \geq x$).

2-1 MUX

Here's the behavior of the 2-1 MUX, abstractly.



If $c == 0$, then x_0 is directed to the output z . If $c == 1$, then x_1 is directed to the output z .

Truth Table for 2-1 MUX

Row	c	x1	x0	z
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

The first four rows have $c = 0$, so they select column x_0 . The second four rows have $c = 1$, so they select column x_1 .

We can shrink the number of rows by creating a condensed truth table

Row	c	z
0	0	x_0
1	1	x_1

The control bit is considered as a 1 bit number ($c = 0$ or $c = 1$). The control bit selects the input with the same value as the control bit. Thus, when $c = 0$, x_0 is selected. When $c = 1$, x_1 is selected.

Row 0: $\neg c x_0$

Row 1: $c x_1$

The minterm for row 0 is normally $\neg c$, but we AND that with the output x_0 to get $\neg c x_0$.

Then, we OR the two modified minterms to get the output:

$$z = \neg c x_0 + c x_1$$

When $c = 0$:

$$z = \neg c x_0 + c x_1 = 1 x_0 + 0 x_1 = x_0$$

When $c = 1$:

$$z = \neg c x_0 + c x_1 = 0 x_0 + 1 x_1 = x_1$$

4-1 MUX

Attributes:

- Data inputs: 4 (x_3, x_2, x_1, x_0)
- Control inputs: 2 (c_1, c_0)
- Outputs: 1 (z)

When $c_1 c_0 = 00$, select x_0 (00 is 0 in base 10).

When $c_1 c_0 = 01$, select x_1 (01 is 1 in base 10).

When $c_1 c_0 = 10$, select x_2 (10 is 2 in base 10).

When $c_1 c_0 = 11$, select x_3 (11 is 3 in base 10).

Truth Table for 4-1 MUX

Row	c1	c0	z
0	0	0	x0
1	0	1	x1
2	1	0	x2
3	1	1	x3

Row 0: $\bar{c}_1\bar{c}_0x_0$

Row 1: $\bar{c}_1c_0x_1$

Row 2: $c_1\bar{c}_0x_2$

Row 3: $c_1c_0x_3$

The Boolean expression for a 4-1 MUX is:

$$z = \bar{c}_1\bar{c}_0x_0 + \bar{c}_1c_0x_1 + c_1\bar{c}_0x_2 + c_1c_0x_3$$

Demultiplexes

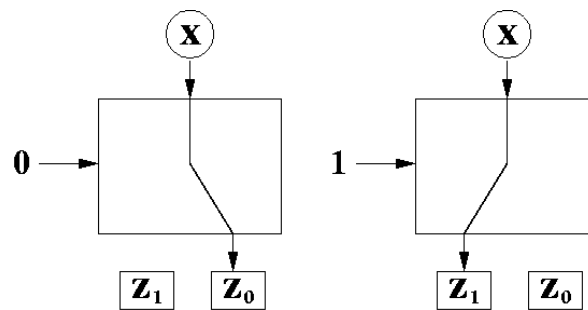
Demultiplexers (DeMUX) are basically multiplexers where the inputs and outputs have been switched.

A 1-n DeMUX consists of the following:

- □ Data inputs: 1
- □ Control inputs: $\text{ceil}(\log_2 n)$
- □ Outputs: n

In a DeMUX, there is a single input, one one of n outputs to choose from to direct the input.

1-2 DeMUX



When $c == 0$, the input x is directed to the output z_0 .

When $c == 1$, the input x is directed to the output z_1 .

Truth Table for 2-1 DeMUX

Row	c	z0	z1
0	0	x	0
1	1	0	x

$$z_1 = c x$$

$$z_0 = \neg c x$$

Encoders and decoders

Decoder

The key features of the decoder. How many data inputs, control inputs, and outputs it has:

a k - $\text{ceil}(\lg(k))$ decoder.

- □ Data inputs: $\text{ceil}(\lg k)$ (labeled $x_{\text{ceil}(\lg k)}, \dots, x_0$)
- □ Outputs: k (labeled z_{k-1}, \dots, z_0)

2-4 decoder:

Attributes:

- □ Data inputs: 2 (labeled x_1, x_0)
- □ Outputs: 4 (labeled z_3, \dots, z_0)

For each of the 4 possible inputs, exactly one of the outputs is set to 1. The rest have a value of 0. Thus, if $x_1x_0 = 11$, then $z_3 = 1$ while all other outputs are 0.

Truth Table for 2-4 decoder.

Truth Table for 2-4 decoder.

x_1x_0	Operation
00	$z_0 = 1$
01	$z_1 = 1$
10	$z_2 = 1$
11	$z_3 = 1$

We can write the Boolean expression for each of the outputs. Since this is a conventional truth table, we can write sum-of-products for each output.

$$z_3 = x_1x_0$$

$$z_2 = x_1\bar{x}_0$$

$$z_1 = \bar{x}_1x_0$$

$$z_0 = \bar{x}_1\bar{x}_0$$

3-8 decoder:

Attributes:

- ▫ Data inputs: 3 (labeled x_2, x_1, x_0)
- ▫ Outputs: 8 (labeled z_7, \dots, z_0)

For each of the 8 possible inputs, exactly one of the outputs is set to 1. The rest have a value of 0. Thus, if $x_2x_1x_0 = 011$, then $z_3 = 1$ while all other outputs are 0.

Truth Table for a 3-8 decoder.

$x_2x_1x_0$	Operation
000	$z_0 = 1$
001	$z_1 = 1$
010	$z_2 = 1$
011	$z_3 = 1$
100	$z_4 = 1$
101	$z_5 = 1$
110	$z_6 = 1$
111	$z_7 = 1$

To see what this is doing, let's look at one of the rows of the table above. In particular, look at the last row. When $x_2x_1x_0 = 111$ (that is, $x_2 = 1, x_1 = 1,$ and $x_0 = 1$), then we know 111 is UB for 710.

Thus, we make $z_7 = 1$. All other outputs are set to 0.

Decoder with enable

If the enable is active, it behaves as a regular decoder. If it's not active, then all outputs are 0.

Here's how the enable bit works:

Enable	Operation
$e = 0$	All outputs are 0
$e = 1$	Acts like regular decoder without enable

Encoder

With the 3-8 decoder, the inputs were a 3 bits UB bitstring which told us which output to set high. So, now that inputs are outputs, the output of an 8-3 decoder has 3 bits as output. This should indicate which input was set to 1, and output the binary representation (in UB) of the input. Thus, if $x_5 = 1$ (and all other inputs are 0), then the output is $z_2z_1z_0 = 101$, since 101 is interpreted as 5 in UB.

Thus, we can write a simplified truth table with only those 8 rows.

Input Variable Has Value 1	z_2	z_1	z_0
x_0	0	0	0
x_1	0	0	1
x_2	0	1	0
x_3	0	1	1
x_4	1	0	0
x_5	1	0	1
x_6	1	1	0
x_7	1	1	1

Thus, if $x_3 = 1$, then the 3 output bits, $z_2z_1z_0 = 011$, since 011 is maps to value 3 in UB.

Suppose we wanted to write the Boolean expression for z_2 . As before, we identify the rows with 1's as outputs, and create a minterm.

If we do this, we will have very large minterms.

$$\begin{aligned} z_2 = & \neg x_0 \neg x_1 \neg x_2 \neg x_3 x_4 \neg x_5 \neg x_6 \neg x_7 + \\ & \neg x_0 \neg x_1 \neg x_2 \neg x_3 \neg x_4 x_5 \neg x_6 \neg x_7 + \\ & \neg x_0 \neg x_1 \neg x_2 \neg x_3 \neg x_4 \neg x_5 x_6 \neg x_7 + \\ & \neg x_0 \neg x_1 \neg x_2 \neg x_3 \neg x_4 \neg x_5 \neg x_6 x_7 \end{aligned}$$

The terms need not be this large. We've assumed that exactly one input is a 1. This is a strong assumption that allows us to simplify the expression greatly.

$$z_2 = x_4 + x_5 + x_6 + x_7$$

$$z_1 = x_2 + x_3 + x_6 + x_7$$

$$z_0 = x_1 + x_3 + x_5 + x_7$$

Sequential Circuits

State

There is a generic definition of state:

State is a quantity that stores the previous history of what's happened in such a way that you can predict the output, given any future input.

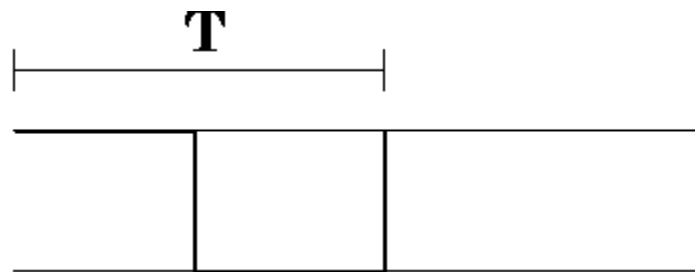
From the definition, we can get two things. First, state refers to some recorded information. We can consider state as a kind of memory. For sequential logic circuits, this is some finite number of bits. Second, there is always a notion of time when we talk about state. Usually, we think of time being involved with state, because it may change over time.

Let's think of an object, it starts off in an initial state, and over time, that state changes. For example, if the object is a piece of paper, when we are drawing we may draw or erase something (add or remove elements) over time. Our picture's state may not keep track of all the things that have happened to it, nevertheless, the picture changes from state to state, as various operations are performing object.

Clock

A clock is an important device in sequential logic that alternates between 0 and 1 repeatedly. Sequential logic circuits, implement functions with state. That is, they keep information internally. The output of a sequential circuit depends not only on the input bits, but also on the internal state. It turns out to be easier for sequential logic to design with a clock. For one thing, the system can change state automatically. For another, the time a state lasts can be controlled by a clock.

The most important feature is the amount of time it takes before the signal repeats. This time is called the period, which we call T . In this period, there is a single cycle. Look at one cycle of the clock. In this one cycle, the clock has an output of 1 for part of the time, and 0 for part of the time.



Flip-Flop

The basic building blocks of sequential logic circuits are flip flops which are devices that can store message. Basically, a real flip flop has two inputs. One input is a control input. For a T flip flop, the control input is labelled T. The other input is the clock. Positive edge-triggered flip flops can only change output values when the clock is at a positive edge. While negative edge triggered flip flops change on a negative edge, and level-triggered flip flops, that change only when the value is 1. When the clock is not at a positive edge, then the output value is held. That is, it does not change. A flip flop's output is the bit it stored. Thus, the flip flop is always outputting the one bit of information.

T flip flop

Here's the characteristic table for a T flip flop.

T	Q	Q ⁺	Operation
0	0	0	Hold
0	1	1	Hold
1	0	1	Toggle
1	1	0	Toggle

The T flip flop characteristic table has 3 columns. The first column is the value of T, a control input. The second column is the current state, that is the current value being output by Q. The third column is the next state, that is, the value of Q at the next positive edge. It's labelled with Q and the superscript, + (the plus sign).

The T flip flop has two possible values. When T = 0, the flip flop does a hold. A hold means that the output, Q is kept the same. When T = 1, the flip flop does a toggle, which means the output Q is negated. Thus, in a T flip flop, you can either maintain the current state's value for another cycle, or you can toggle the value (negate it).

Why isn't it XOR?

First, XOR has two inputs, and one output. A T flip flop essentially has a control input and a control output. Second, the second column and the third column are really the same output, but at different points in time.

D flip flop

Here's the characteristic table for a D flip flop.

D	Q	Q+	Operation
0	0	0	Reset
0	1	0	Reset
1	0	1	Set
1	1	1	Set

The D flip flop characteristic table has 3 columns. The first column is the value of D, a control input. The second column is the current state, that is the current value being output by Q. The third column is the next state, that is, the value of Q at the next positive edge. It's labelled with Q and the superscript, + (the plus sign). Sometimes, the current state is written as Q(t) which means the value of Q at the current time, t, and the next state is written as Q(t + 1) which means the value of Q at the next clock edge. However, I'll usually write it as Q+.

In the characteristic table, the second column isn't really an input, it's an output. The third column is really the same output, but just the output at a future time. The D flip flop has two possible values. When D = 0, the flip flop does a reset. A reset means that the output, Q is set to 0. When D = 1, the flip flop does a set, which means the output Q is set to 1. When the clock is not at a positive edge, the flip flop ignores D. However, at the positive edge, it reads in the value, D, and based on D, it updates the value of Q (and of course, Q').

There is some small amount of delay while it reads in the control input (from D) and the output. In fact, the "D" in D flip flop stands for "delay". It basically means that the "D" value is not read immediately, but only at the next positive clock edge.

JK flip flop

Here's the characteristic table for a JK flip flop.

J	K	Q	Q+	Operation
0	0	0	0	Hold
0	0	1	1	Hold
0	1	0	0	Reset
0	1	1	0	Reset
1	0	0	1	Set
1	0	1	1	Set

1	1	0	1	Toggle
1	1	1	0	Toggle

Basically, a JK flip flop is a combination of a D and T flip flop (or more accurately, a D and T flip flop are a simplification of a JK flip flop). A JK flip flop has two control inputs, J and K. When JK = 00, the flip flop holds. When JK = 01, the flip flop resets. When JK = 10, the flip flop sets. When JK = 11, the flip flop toggles.

For the most part, we'll ignore JK flip flops, and JK flip flops are only mentioned for completeness.

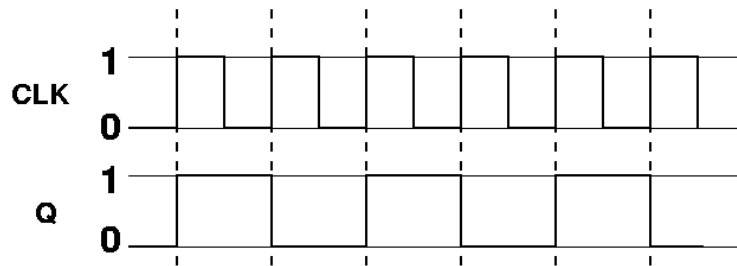
Difference between Flip flop and gate

The biggest difference between a flip flop and a gate is that a flip flop can hold its value. Even though holding a value is something very simple, it makes it different from logic gates, and allows us to design circuits that have cycles in them (i.e., feedback).

Asynchronous Counters

An asynchronous counter is a device that counts from 0 to N without all flip flops hooked to the same clock. We are going to build one using T flip flops.

If the control input of a T flip flop input is 1, we can observe the behavior of the output of a T flip flop by looking at a timing diagram.



The output, Q, resembles a clock as well. If the period of the clock is T, then the period of Q (the output of the flip) is 2T.

If we feed the clock into a T flip flop, where T is hardwired to 1, the output will be a clock whose period is twice as long. If we feed the output of this T flip flop, whose period is 2T, as the clock of another T flip flop, which also has its T input hardwired to 1. It creates a clock that has twice the period, the output of the second flip flop has period 4T.

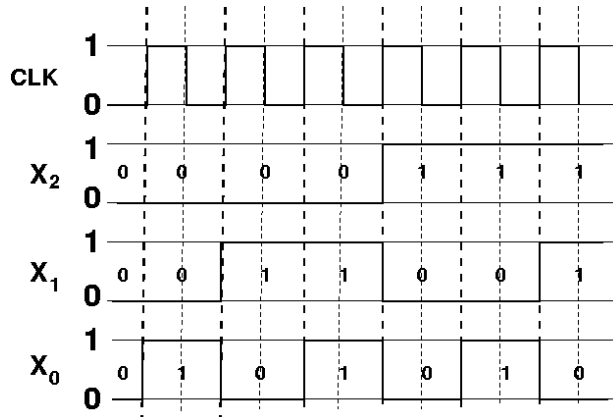
If keep feeding the output of one T flip flop into the clock input of another T flip flop hardwired to 1, the period of the output of the Nth flip flop is $2^N T$. Each flip flop doubles the period, so N flip flops is 2 raised to the Nth power.

Row	x2	x1	x0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1

6	1	1	0
7	1	1	1

Look at the column labelled x_0 . It reads 0, 1, 0, 1, 0, 1, etc. It looks like a clock. Assume that this clock has period T . Now look at the column labelled x_1 . It reads 0, 0, 1, 1, 0, 0, 1, 1. It looks like a clock too. However, it stays 0 twice as long, then 1 twice as long. In fact, it looks like a clock that has a period of $2T$. Now look at the column labelled x_2 . It reads 0, 0, 0, 0, 1, 1, 1, 1. Again, this looks like a clock, except it's going twice as slow. It has a period of $4T$. They look like the chained T flip flops we had above.

This is the timing diagram that shows how the counter behaves.



First look at the row that says CLK. That's the clock.

Then look at row X_0 . This toggles between 0 and 1 on the positive edge of the clock. That's because the clock is fed into the bottommost T flip flop. X_1 toggles according to X_0 . That's because we feed X_0' into the clock of the middle T flip flop. X_2 toggles according to X_1 . That's because we feed X_1' into the clock of the top T flip flop.

Start reading the timing diagram from the left most column, you will see 000, then 001, then 010, then 011, and so forth. As you can see the output of the flip flops is incrementing as it should. The counter increments at a period of $2T$, assuming the clock has a period of T .

Creating an asynchronous counter from T flip flops relies on two observations. First, if you hardwire a 1 into a T flip flop, the output of the T flip flop (i.e., Q) toggles at twice the period.

Second, if x_0 acts like a clock of period T, then x_i acts like a clock of period $2^i T$. That is, each success column to the left is a clock that doubles the period. Thus, we can combine these two facts together to generate a counter. Notice that the counter must increment based on negative edges. Thus, X_{i+1} toggles on X_i . This is accomplished by feeding the negative output of a T flip flop (i.e. Q) to the clock of the next T flip flop.

This counter is considered asynchronous, since each flip flop runs on its own clock.