

How Boolean Logic Works

by Marshall Brain

Browse the article [How Boolean Logic Works](#)



Boolean logic affects how computers operate. HowStuffWorks.com

Introduction to How Boolean Logic Works

Have you ever wondered how a computer can do something like balance a check book, or [play chess](#), or spell-check a document? These are things that, just a few decades ago, only humans could do. Now computers do them with apparent ease. How can a "chip" made up of silicon and wires do something that seems like it requires human thought?

If you want to understand the answer to this question down at the very core, the first thing you need to understand is something called **Boolean logic**. Boolean logic, originally developed by George Boole in the mid 1800s, allows quite a few unexpected things to be mapped into [bits and bytes](#). The great thing about Boolean logic is that, once you get the hang of things, Boolean logic (or at least the parts you need in order to understand the operations of computers) is outrageously simple. In this article, we will first discuss simple logic "gates," and then see how to combine them into something useful.

Simple Gates



NOT Gate

There are three, five or seven simple gates that you need to learn about, depending on how you want to count them (you will see why in a moment). With these simple gates you can build combinations that will implement any digital component you can imagine. These gates are going to seem a little dry here, and incredibly simple, but we will see some interesting combinations in the following sections that will make them a lot more inspiring. If you have not done so already, reading [How Bits and Bytes Work](#) would be helpful before proceeding.

NOT Gate

The simplest possible gate is called an "inverter," or a NOT gate. It takes one bit as input and produces as output its opposite. The logic table is:

A	Q
0	1
1	0

AND Gate



The NOT gate has one input called **A** and one output called **Q** ("Q" is used for the output because if you used "O," you would easily confuse it with zero). The table shows how the gate behaves. When you apply a 0 to A, Q produces a 1. When you apply a 1 to A, Q produces a 0. Simple.

OR Gate

AND Gate

The AND gate performs a logical "and" operation on two inputs, A and B:

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

NOR Gate

1 0 0

1 1 1

The idea behind an AND gate is, "If A **AND** B are both 1, then Q should be 1." You can see that behavior in the logic table for the gate. You read this table row by row, like this:

- A B Q**
- 0 0 0 *If A is 0 AND B is 0, Q is 0.*
- 0 1 0 *If A is 0 AND B is 1, Q is 0.*
- 1 0 0 *If A is 1 AND B is 0, Q is 0.*
- 1 1 1 *If A is 1 AND B is 1, Q is 1.*

OR Gate

The next gate is an OR gate. Its basic idea is, "If A is 1 **OR** B is 1 (or both are 1), then Q is 1."

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

1 0 1

1 1 1



NAND Gate

Those are the three basic gates (that's one way to count them). It is quite common to recognize two others as well: the **NAND** and the **NOR** gate. These two gates are simply combinations of an AND or an OR gate with a NOT gate. If you include these two gates, then the count rises to five. Here's the basic operation of NAND and NOR gates -- you can see they are simply inversions of AND and OR gates:

NOR Gate

A B Q

0 0 1

0 1 0

1 0 0

1 1 0

NAND Gate

A B Q

0 0 1

0 1 1

1 0 1

1 1 0

The final two gates that are sometimes added to the list are the **XOR** and **XNOR** gates, also known as "exclusive or" and "exclusive nor" gates, respectively. Here are their tables:

XOR Gate

A B Q

0 0 0

0 1 1

1 0 1

1 1 0

XNOR Gate

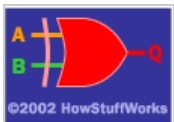
A B Q

0 0 1

0 1 0

1 0 0

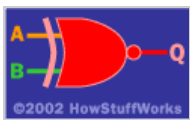
1 1 1



XOR Gate

The idea behind an XOR gate is, "If either A **OR** B is 1, but **NOT** both, Q is 1." The reason why XOR might not be included in a list of gates is because you can implement it easily using the original three gates listed.

If you try all four different patterns for A and B and trace them through the circuit, you will find that Q behaves like an XOR gate. Since there is a well-understood symbol for XOR gates, it is generally easier to think of XOR as a "standard gate" and use it in the same way as AND and OR in circuit diagrams.



XNOR Gate

Simple Adders

In the article on [bits and bytes](#), you learned about **binary addition**. In this section, you will learn how you can create a circuit capable of binary addition using the gates described in the previous section.

Let's start with a **single-bit adder**. Let's say that you have a project where you need to add single bits together and get the answer. The way you would start designing a circuit for that is to first look at all of the logical combinations. You might do that by looking at the following four sums:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

That looks fine until you get to $1 + 1$. In that case, you have that pesky **carry bit** to worry about. If you don't care about carrying (because this is, after all, a 1-bit addition problem), then you can see that you can solve this problem with an XOR gate. But if you do care, then you might rewrite your equations to always include **2 bits of output**, like this:

$$0 + 0 = 00$$

$$0 + 1 = 01$$

$$1 + 0 = 01$$

$$1 + 1 = 10$$

From these equations you can form the logic table:

1-bit Adder with Carry-Out

A B Q CO

0 0 0 0

0 1 1 0

1 0 1 0

1 1 0 1

By looking at this table you can see that you can implement Q with an XOR gate and CO (carry-out) with an AND gate. Simple.

What if you want to add two 8-bit bytes together? This becomes slightly harder. The easiest solution is to modularize the problem into **reusable components** and then replicate components. In this case, we need to create only one component: a **full binary adder**.

The difference between a full adder and the previous adder we looked at is that a full adder accepts an A and a B input plus a **carry-in** (CI) input. Once we have a full adder, then we can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

In the next section, we'll look at how a full adder is implemented into a circuit.

Full Adders

The logic table for a full adder is slightly more complicated than the tables we have used before, because now we have **3 input bits**. It looks like this:

One-bit Full Adder with Carry-In and Carry-Out

CI A B Q CO

0 0 0 0 0

0 0 1 1 0

0 1 0 1 0

0 1 1 0 1

1 0 0 1 0

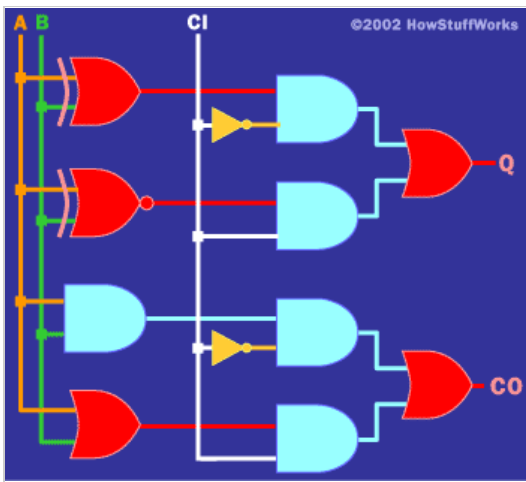
1 0 1 0 1

1 1 0 0 1

1 1 1 1 1

There are many different ways that you might implement this table. I am going to present one method here that has the benefit of being easy to understand. If you look at the Q bit, you can see that the top 4 bits are behaving like an XOR gate with respect to A and B, while the bottom 4 bits are behaving like an XNOR gate with respect to A and B. Similarly, the top 4 bits of CO are behaving like an AND gate with respect to A and B, and the bottom 4 bits behave like an OR gate. Taking those facts, the following circuit implements a full adder:

This definitely is not the most efficient way to implement a full adder, but it is extremely easy to understand and trace through the logic using this method. If you are so inclined, see what you can do to implement this logic with fewer gates.



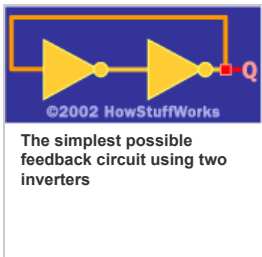
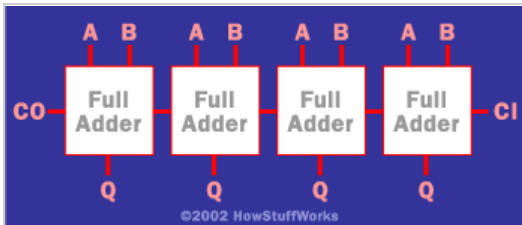
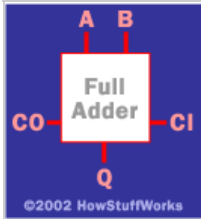
Full adders can be implemented in a wide variety of ways.

Now we have a piece of functionality called a "full adder." What a computer engineer then does is "black-box" it so that he or she can stop worrying about the details of the component. A **black box** for a full adder would look like this:

With that black box, it is now easy to draw a **4-bit full adder**:

In this diagram the carry-out from each bit feeds directly into the carry-in of the next bit over. A 0 is hard-wired into the initial carry-in bit. If you input two 4-bit numbers on the A and B lines, you will get the 4-bit sum out on the Q lines, plus 1 additional bit for the final carry-out. You can see that this chain can extend as far as you like, through 8, 16 or 32 bits if desired.

The 4-bit adder we just created is called a **ripple-carry** adder. It gets that name because the carry bits "ripple" from one adder to the next. This implementation has the advantage of simplicity but the disadvantage of speed problems. In a real circuit, gates take time to switch states (the time is on the order of nanoseconds, but in high-speed computers nanoseconds matter). So 32-bit or 64-bit ripple-carry adders might take 100 to 200 nanoseconds to settle into their final sum because of carry ripple. For this reason, engineers have created more advanced adders called **carry-lookahead** adders. The number of gates required to implement carry-lookahead is large, but the settling time for the adder is much better.



The simplest possible feedback circuit using two inverters

Flip Flops

One of the more interesting things that you can do with Boolean gates is to create **memory** with them. If you arrange the gates correctly, they will remember an input value. This simple concept is the basis of **RAM** (random access memory) in computers, and also makes it possible to create a wide variety of other useful circuits.

Memory relies on a concept called **feedback**. That is, the output of a gate is fed back into the input. The simplest possible feedback circuit using two inverters is shown above.

If you follow the feedback path, you can see that if Q happens to be 1, it will always be 1. If it happens to be 0, it will always be 0. Since it's nice to be able to control the circuits we create, this one doesn't have much use -- but it does let you see how feedback

works.

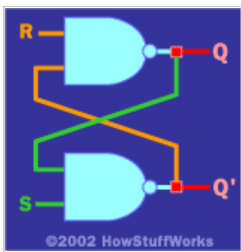
It turns out that in "real" circuits, you can actually use this sort of simple inverter feedback approach. A more useful **feedback circuit using two NAND gates is shown below**:

This circuit has two inputs (**R** and **S**) and two outputs (**Q** and **Q'**). Because of the feedback, its logic table is a little unusual compared to the ones we have seen previously:

R	S	Q	Q'
0	0	Illegal	
0	1	1	0
1	0	0	1
1	1	Remembers	

What the logic table shows is that:

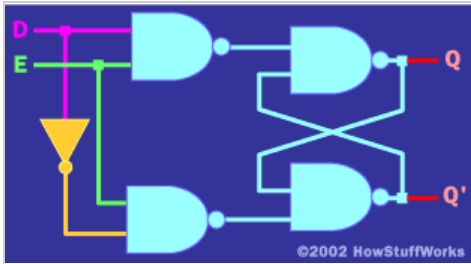
- If R and S are opposites of one another, then Q follows S and Q' is the inverse of Q.
- If both R and S are switched to 1 simultaneously, then the circuit **remembers** what was previously presented on R and S.



There is also the funny **illegal** state. In this state, R and S both go to 0, which has no value in the memory sense. Because of the illegal state, you normally add a little **conditioning logic** on the input side to prevent it, as **shown here**:

In this circuit, there are two inputs (D and E). You can think of **D** as "Data" and **E** as "Enable." If E is 1, then Q will follow D. If E changes to 0, however, Q will remember whatever was last seen on D. A circuit that behaves in this way is generally referred to as a **flip-flop**.

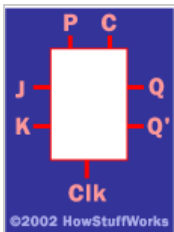
In the next section we'll look at the J-K flip-flop.



The J-K Flip-Flop

A very common form of flip-flop is the **J-K flip-flop**. It is unclear, historically, where the name "J-K" came from, but it is generally represented in a black box like this:

In this diagram, **P** stands for "Preset," **C** stands for "Clear" and **Clk** stands for "Clock."

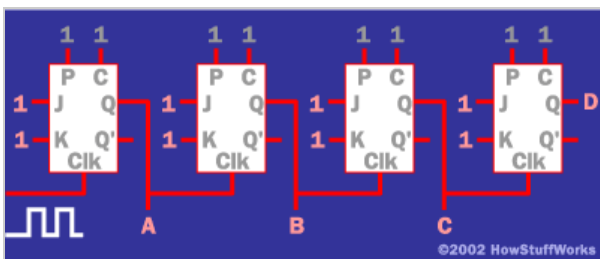


P	C	Clk	J	K	Q	Q'
1	1	1-to-0	1	0	1	0
1	1	1-to-0	0	1	0	1
1	1	1-to-0	1	1	Toggles	
1	0	x	x	x	0	1
0	1	x	x	x	1	0

J-K flip-flop can be used to create and edge-triggered latch, which is important to the design of CPUs.

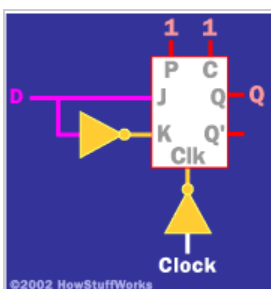
Here is what the table is saying: First, Preset and Clear override J, K and Clk completely. So if Preset goes to 0, then Q goes to 1; and if Clear goes to 0, then Q goes to 0 no matter what J, K and Clk are doing. However, if both Preset and Clear are 1, then J, K and Clk can operate. The **1-to-0** notation means that when the clock changes from a 1 to a 0, the value of J and K are remembered if they are opposites. At the **low-going edge** of the clock (the transition from 1 to 0), J and K are stored. However, if both J and K happen to be 1 at the low-going edge, then Q simply **toggles**. That is, Q changes from its current state to the opposite state.

You might be asking yourself right now, "What in the world is that good for?" It turns out that the concept of "edge triggering" is very useful. The fact that J-K flip-flop only "latches" the J-K inputs on a transition from 1 to 0 makes it much more useful as a memory device. J-K flip-flops are also extremely useful in **counters** (which are used extensively when **creating a digital clock**). Here is an example of a **4-bit counter using J-K flip-flops**:

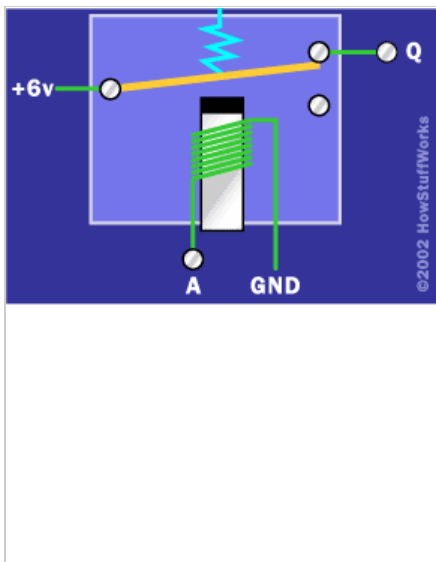


The outputs for this circuit are A, B, C and D, and they represent a 4-bit binary number. Into the clock input of the left-most flip-flop comes a signal changing from 1 to 0 and back to 1 repeatedly (an **oscillating signal**). The counter will count the low-going edges it sees in this signal. That is, every time the incoming signal changes from 1 to 0, the 4-bit number represented by A, B, C and D will increment by 1. So the count will go from 0 to 15 and then cycle back to 0. You can add as many bits as you like to this counter and count anything you like. For example, if you put a magnetic switch on a door, the counter will count the number of times the door is opened and closed. If you put an optical sensor on a road, the counter could count the number of cars that drive by.

Another use of a J-K flip-flop is to create an **edge-triggered latch**, as shown here.



In this arrangement, the value on D is "latched" when the clock edge goes from low to high. **Latches** are extremely important in the design of things like **central processing units (CPUs)** and peripherals in computers.



Implementing Gates

In the previous sections we saw that, by using very simple Boolean gates, we can implement adders, counters, latches and so on. That is a big achievement, because not so long ago human beings were the only ones who could do things like add two numbers together. With a little work, it is not hard to design Boolean circuits that implement subtraction, multiplication, division... You can see that we are not that far away from a pocket calculator. From there, it is not too far a jump to the full-blown CPUs used in computers.

So how might we implement these gates in real life? Mr. Boole came up with them on paper, and on paper they look great. To use them, however, we need to implement them in physical reality so that the gates can perform their logic actively. Once we make that leap, then we have started down the road toward creating real computation devices.

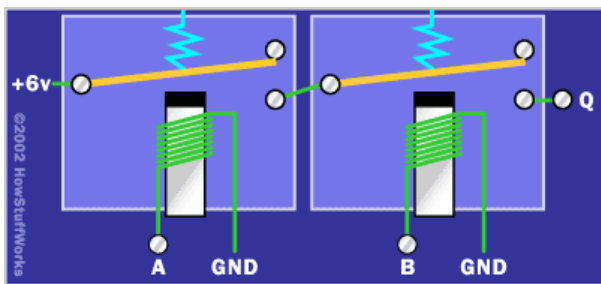
The easiest way to understand the physical implementation of Boolean logic is to use relays. This is, in fact, how the very first computers were implemented. No one implements computers with relays anymore -- today, people use sub-microscopic transistors etched onto silicon chips. These transistors are incredibly small and fast, and they consume very little power compared to a relay. However, relays are incredibly easy to understand, and they can implement Boolean logic very simply. Because of that simplicity, you will be able to see that mapping from "gates on paper" to "active gates implemented in physical reality" is possible and straightforward. Performing the same mapping with transistors is just as easy.

Let's start with an inverter. Implementing a NOT gate with a relay is easy: What we are going to do is use voltages to represent bit states. We will define a binary 1 to be 6 volts and a binary 0 to be zero volts (ground). Then we will use a 6-volt battery to power our circuits. Our NOT gate will therefore look like this:

[If this figure makes no sense to you, please read [How Relays Work](#) for an explanation.]

You can see in this circuit that if you apply zero volts to A, then you get 6 volts out on Q; and if you apply 6 volts to A, you get zero volts out on Q. It is very easy to implement an inverter with a relay!

It is similarly easy to implement an AND gate with two relays:



Here you can see that if you apply 6 volts to A and B, Q will have 6 volts. Otherwise, Q will have zero volts. That is exactly the behavior we want from an AND gate. An OR gate is even simpler -- just hook two wires for A and B together to create an OR. You can get fancier than that if you like and use two relays in parallel.

You can see from this discussion that you can create the three basic gates -- NOT, AND and OR -- from relays. You can then hook those physical gates together using the logic diagrams shown above to create a physical 8-bit ripple-carry adder. If you use simple switches to apply A and B inputs to the adder and hook all eight Q lines to light bulbs, you will be able to add any two numbers together and read the results on the lights ("light on" = 1, "light off" = 0).

Boolean logic in the form of simple gates is very straightforward. From simple gates you can create more complicated functions, like addition. Physically implementing the gates is possible and easy. From those three facts you have the heart of the digital revolution, and you understand, at the core, how computers work.

Lots More Information

Related Articles

- [How Electronic Gates Work](#)
- [How Bits and Bytes Work](#)
- [How Microprocessors Work](#)
- [How Digital Clocks Work](#)
- [How Relays Work](#)
- [How Chess Computers Work](#)

More Great Links

- [Boolean Searching on the Internet](#)
- [Logic Gates and Boolean Algebra](#)
- [PC Mechanic: Boolean Logic and Gates](#)
- [Control And Embedded Systems: Boolean Logic](#)