Digital Circuits and Boolean Logic

Introduction

Digital or binary logic has fascinated many people over the years. The very idea that a two-valued number system can possibly be the basis for the most powerful and sophisticated computers seems astounding, to say the least. Nevertheless, it is so, and the how and the why of this requires some explanation.

Everything in the digital world is based on the binary number system. Numerically, this involves only two symbols: 0 and 1. Logically, we can use these symbols or we can equate them with others according to the needs of the moment. Thus, when dealing with digital logic, we can specify that:

0 = false = no
1 = true = yes

Using this two-valued logic system, every statement or condition must be either "true" or "false;" it cannot be partly true and partly false. While this approach may seem limited, it actually works quite nicely, and can be expanded to express very complex relationships and interactions among any number of individual conditions.

Storing Digital/Binary Values

Information can be represented and stored on a variety of electrical/mechanical devices. In many cases, the information relates to measurable variables such as elapsed time or total rainfall or accumulated electrical charge (for which the hourglass, raingauge, and capacitor, respectively, are suitable representation devices). But what about abstract information, such as quantities in mathematics? Here we create an analogy between something that can be stored and measured in an electrical/mechanical device and a mathematical value. For example, we can assign an equivalence between mathematical value and electrical charge. The extent to which we can operate on that electrical charge via the capacitor and our measuring instruments is the degreee to which we can perform analogous mathematical calculations.

There are problems, however, with relating mathematics to storable parameters on physical devices. Two of the more important ones are 1) measuring instruments are rarely more accurate than three decimal digits--so mathematics carried out through these devices would have intrinsic limited accuracy; 2) there is typically unrecoverable loss of information--a capacitor could leak away part of its charge, i.e., its analogous mathematical value would arbitrarily change.

But there is a solution to these problems: store mathematical values in discrete rather than analog form. Here one uses devices whose variations are limited to discrete states--

typically two, e.g., on or off, positive or negative, closed or open. Then, by representing mathematical quantities in a number system having only two digits--a binary number system--any value can be represented with arbitrary accuracy by linking together a sequence of two-state devices and setting the appropriate state for each device. Information integrity in this discrete representation is better than that of analog representation because here information loss requires an arbitrary change of state of a device, not a drift in value. That is much less likely, and there are ways to correct for it.

Turning 1s and 0s into Meaningful Information: Boolean Logic

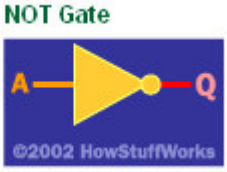
Have you ever wondered how a computer can do something like balance a check book, or play chess, or spell-check a document? These are things that, just a few decades ago, only humans could do. Now computers do them with apparent ease. How can a "chip" made up of silicon and wires do something that seems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

Logic Gates: AND, OR, NOT, NAND, NOR

There are five simple gates that you need to learn about. With these simple gates you can build combinations that will implement any digital component you can imagine.

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 table below shows a logic table for the NOT gate and the normal symbol for it in circuit diagrams:

A	Q
0	1
1	0

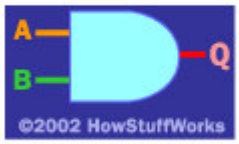


You can see in this figure that the NOT gate has one input called **A** and one output called **Q** ("Q" is used for the output because if you used "O," you would easily confuse it with zero). The table shows how the gate behaves. When you apply a 0 to A, Q produces a 1. When you apply a 1 to A, Q produces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

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

AND Gate

A	B	Q
0	0	0
0	1	0
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:

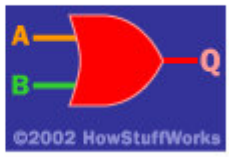
AND Gate

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.*

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."

OR Gate

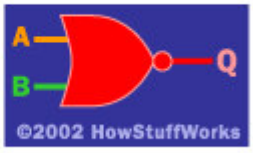
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1



Those are the three basic. 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:

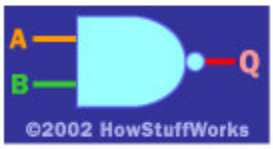
NOR Gate

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



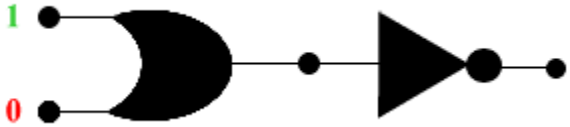
NAND Gate

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

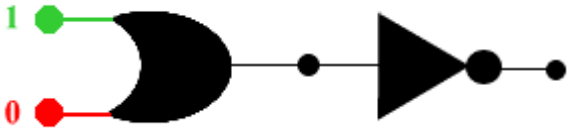


Logic Gates Exercises

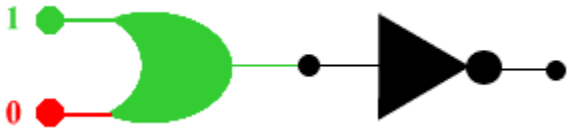
Given some inputs, you can use the diagram of the gates to figure out what the output will be. We'll do one example together and then you'll do a few on your own. Here is our example:



Based on the inputs, we know how the electricity will flow up to the gate:

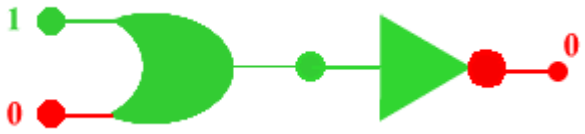


Now, because OR gates let electricity through when at least one of their inputs is on, we can let the electricity flow all the way through the OR gate:



It can now flow freely all the way up to the NOT gate:

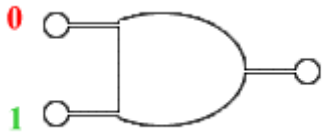
We know that when a NOT gate gets a 1 as input, it puts out a 0 as output:



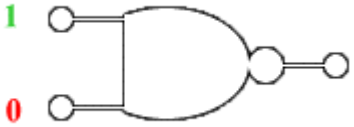
And we are done!

On the following diagrams, indicate whether the output value will be 0 or 1. Use colored pens/pencils (green for on, red for off) to show how electricity flows through the gates:

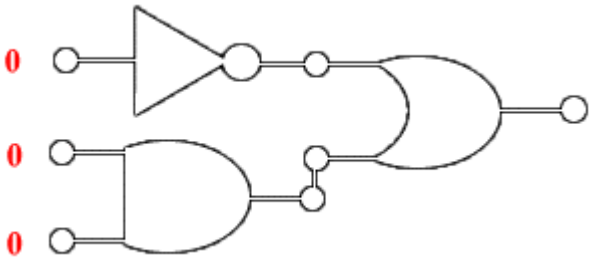
1.



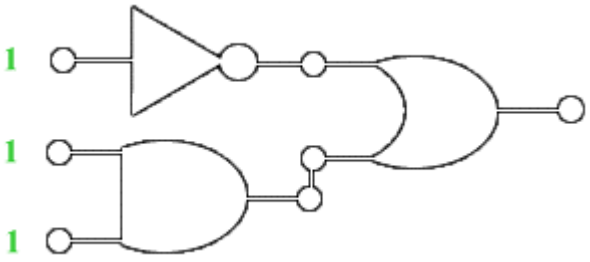
2.



3.



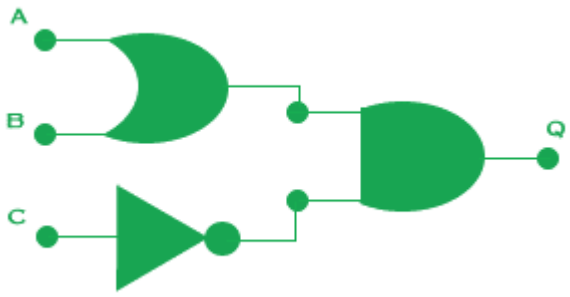
4.



Truth Tables

Truth tables are a way of figuring out how a particular digital circuit will respond to different inputs. Truth tables have two distinct parts: each left hand column is labeled with the name of one of the inputs, each right hand column is labeled by a piece of the circuit whose output is to be determined. To fill in a truth table, you begin with the left-hand columns and make one row for each possible combination of the values for the inputs. Then, for each row of the table, using the values in the left hand side and the logic of the circuit, you can fill in the appropriate values in the right-hand sides.

Let's try an example. Here is the machine we'll work with:



We have 3 inputs. Each input has 2 possible values. So we'll have 2 to the 3d, or 8, different possible combinations of the input values. First we'll make 3 left hand columns, one for each input, and fill each of the rows in with the 8 different possibilities:

A	B	C	
T	T	T	|
T	T	F	|
T	F	T	|
T	F	F	|
F	T	T	|
F	T	F	|
F	F	T	|
F	F	F	|

Note that in the chart above we used T and F instead of 0 and 1. You can consider these to be interchangeable. (T = true = 1 = on, F = false = 0 = off.) Next we can label the rest of the columns for each of the pieces of the machine:

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

Next we can use the information in the left side of each row to help us fill in the rest of the rows. We'll do the first row step by step. First we want to fill in the output of A OR B. To do this we look at the values of A and B for that row. We see that they are both true. We know from the meaning of an OR gate that if either of its inputs is true, then the output will be true. So in this case, we know the output will be true and we can enter that into the truth table:

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

We can use the same process to figure out the output of NOT C for the first row. In the first row, the value of C is T. Because NOT always gives the opposite of its input, we know the output will be false. So we can enter that into the table:

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

Finally, we are ready to figure out the output for the whole machine for the inputs given in the first row. The inputs to the final AND gate are the outputs from the other two gates that we've already figured out: (A OR B) outputs T and (NOT C) outputs F. We know from the meaning of an AND gate that both its inputs must be true for it to output true, so in this case it will output false:

A	B	C		A OR B	NOT C	(A OR B) AND (NOT C)
T	T	T	|	T	F	F
T	T	F	|			
T	F	T	|			
T	F	F	|			
F	T	T	|			
F	T	F	|			
F	F	T	|			
F	F	F	|			

Now we can continue with the same process for the rest of the possibilities for the inputs. Try it yourself and make sure you get the same answers:

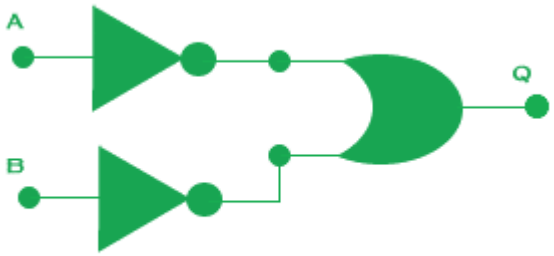
A	B	C		A OR B	NOT C	(A OR B) AND (NOT C)
T	T	T	|	T	F	F
T	T	F	|	T	T	T
T	F	T	|	T	F	F
T	F	F	|	T	T	T
F	T	T	|	T	F	F
F	T	F	|	T	T	T
F	F	T	|	F	F	F
F	F	F	|	F	T	F

Once we have filled in the entire truth table, we know what the output of our circut will be for any possible combination of input values. You can use truth tables to help analyze digital circuits and to help design them.

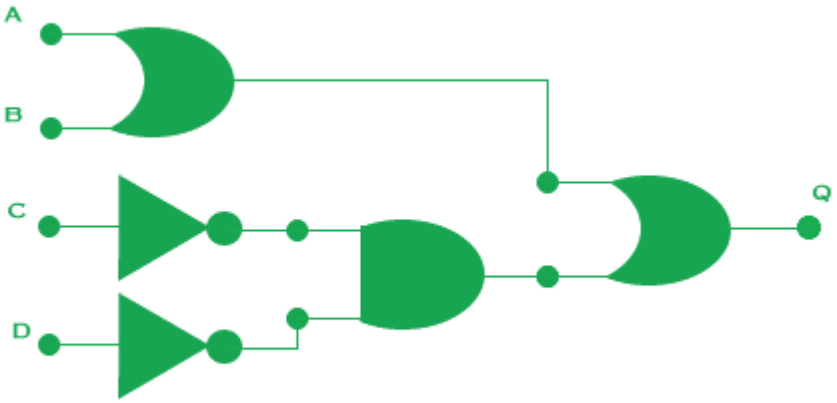
Truth Table Practice Problems

A. Draw and fill in truth tables for the following machines:

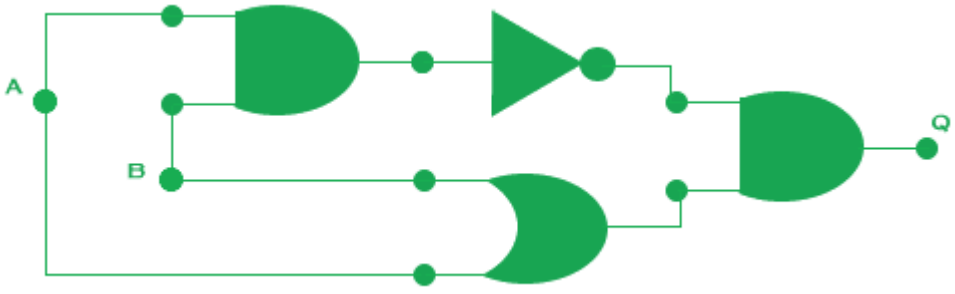
1.



2.



3.



B. Describe what circuit A.3 accomplishes.

C. Build the following circuits and prove their correctness using truth tables:

1. Build a circuit that simulates a NAND gate using only AND, OR, and NOT gates. You need not use all three types of gate in your design. Use the NAND truth table and one for your circuit to prove that your circuit is correct.
2. Build a circuit that simulates a NOR gate using only AND, OR, and NOT gates. You need not use all three types of gate in your design. Use the NOR truth table and the one for your machine to prove that it is correct.
3. Using only a NAND gate, build circuits to simulate AND, OR and NOT gates. Use the AND, OR, and NOT truth tables and the ones from your machine to prove that it is correct.

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.

Relays

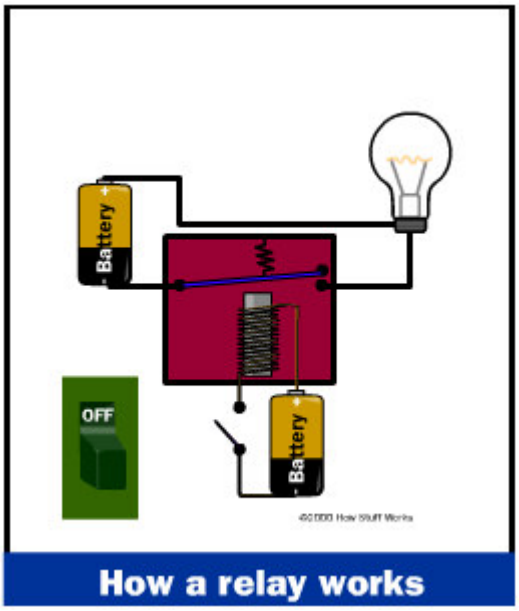
A relay is a simple electromechanical switch made up of an electromagnet and a set of contacts. Relays are found hidden in all sorts of devices. In fact, some of the first computers ever built used relays to implement Boolean gates.

Relay Construction

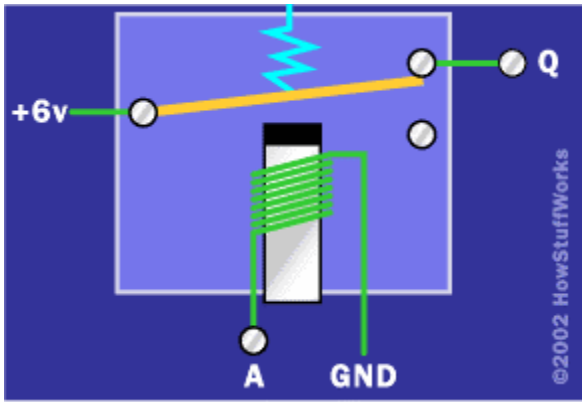
Relays are amazingly simple devices. There are four parts in every relay:

1. Electromagnet
2. Armature that can be attracted by the electromagnet
3. Spring
4. Set of electrical contacts

The following figure shows these four parts in action:

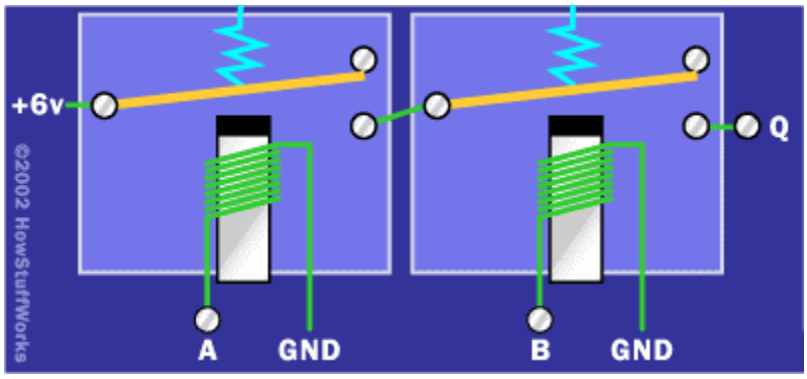


In this figure, you can see that a relay consists of two separate and completely independent circuits. The first is at the bottom and drives the electromagnet. In this circuit, a switch is controlling power to the electromagnet. When the switch is on, the electromagnet is on, and it attracts the armature (blue). The armature is acting as a switch in the second circuit. When the electromagnet is energized, the armature completes the second circuit and the light is on. When the electromagnet is not energized, the spring pulls the armature away and the circuit is not complete. In that case, the light is dark. 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:



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 circuit to do whatever you like.

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.

Rocky's Boots

Quite a long time ago a couple of people wrote a computer game that allows you to build digital circuits to solve problems. The game is called Rocky's Boots and it is both

challenging and fun. Your teacher may have access to this program in your classroom. If you have additional time for extra challenge work, ask him/her if it is available and see how many levels you can get through!

Bibliography

This worksheet is largely excerpted and edited from articles by Marshall Brain on the How Stuff Works website: <http://www.howstuffworks.com>

Brain, Marshall. "How Stuff Boolean Logic Works".
Brain, Marshall. "How Electronic Gates Work".
Brain, Marshall. "How Relays Work".

Additional information is used from:

"Digital Circuits", Johns Hopkins Virtual Laboratories, <http://www.jhu.edu/~virtlab/>.
"Digital Logic" at <http://www.play-hookey.com>.