Digital Electronics

Binary Number System

Before introducing the binary (base 2) number system, let's review the decimal (base 10) number system. The base 10 system is the counting system that we use everyday. There are ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We can represent any number in base 10 by listing the digits such that the rightmost digit tells us the number of "ones" (10^0). As we move left, the digits tell us how many "tens" (10^1), "hundreds" (10^2), etc., are in the number. We add up all of these "ones", "tens", "hundreds", etc., to get the final number. For example, we represent the number "one hundred fifty-three" numerically in base 10 as "153" since $1*10^2 + 5*10^1 + 3*10^0 = 153$.

A base 2 number system works similarly. In this case there are only two digits: 0 and 1. As with base 10, reading from right to left, the digits correspond to the number of 2^0's, 2^1's, 2^2's, etc. For example, we represent the number "one hundred fifty-three" numerically in base 2 as "10011001".

We can convert between the base 2 and base 10 representations using simple arithmetic. In base 2, we saw one hundred fifty-three was represented as 1001001; let's check to make sure this is correct. Reading from right to left, the digits tell us how many 2^0's, 2^1's, 2^2's, etc., are in the number. By adding all of these powers of 2 together, we can represent a base 2 number in base 10. Taking the example of one hundred fifty-three:

$$1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 153.$$

The arithmetic looks cleaner if we only consider the nonzero digits:

$$1*2^7 + 1*2^4 + 1*2^3 + 1*2^0 = 153.$$

Converting from base 10 to base 2 is a little trickier, but still only involves arithmetic. The basic process consists of successively subtracting the largest power of two from the number until you reach zero, while being sure to keep track of which powers of two you've used. Again, let's look at the base 10 numeral 153. The largest power of two that is smaller than 153 is 2^7 (=128), thus we'll put a 1 in the 2^7 column. Subtracting 128 from 153 gives 25. The largest power of two that is smaller than 25 is 2^4 (=16). Since we didn't need 2^6 or 2^5, we'll put 0 in these columns followed by a 1 in the 2^4 column. Continuing this process, we'll need a 1 in the 2^3 column, 0's in the 2^2 and 2^1 columns, and a 1 in the 2^0 column. Writing all of the digits together gives 10011001.

Binary Arithmetic

Let's consider adding in base 2 and make an analogy to adding in base 10. Say, we'd like to add two base 10 numbers together. We know we start by adding the ones digits first, then move to the tens digits, and so on. What happens if two digits add up to more than nine? Since "9" is the largest digit in the base 10 system, we have to "carry the one" over to the next column; this same rule applies in binary addition. The basic rules for adding in binary are:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10 \text{ (remember, “carry the one”)}$$

Example: Add 1001 and 101 in binary

$$\begin{array}{r} 100^{1}1 \\ +\,101 \\ \hline 1110 \end{array}$$

Does this make sense? An easy way to check is to convert the numbers into base 10 and add them. We know 1001 in binary is 9 in base 10, 101 is 5, and 1110 is 14; 9 and 5 is definitely 14, so our answer is correct.

Multiplication in base 2 follows the same algorithm as in base 10. For example, how do we find the product of 13 and 12 in base 10?

$$\begin{array}{r} 13 \\ \times\,12 \\ \hline 26 \\ +\,130 \\ \hline 156 \end{array}$$

This algorithm basically boils down to invoking the distributive property of multiplication—(13 x 12) = (13 x 2) + (13 x 10). Briefly, we multiply 13 by the ones digit of 12 (which is 2) to get 26. On the next line, we add a zero as a placeholder and multiply 13 by the tens digit of 12. (*NB:* The zero placeholder is introduced because, although we explicitly only multiply by 1, we are essentially multiplying by 10.) The zero placeholder is also used in binary multiplication. The basic multiplication rules in binary are:

$$0 \times 0 = 0$$
$$1 \times 0 = 0$$
$$0 \times 1 = 0$$

1 x 1 = 1

Example: Multiply 111 by 11 in binary

$$\begin{array}{r} 111 \\ \times\,11 \\ \hline 111 \\ +\,1110 \\ \hline 10101 \end{array}$$

We can check our arithmetic by converting to decimal: 111 in binary is 7 in decimal, 11 is 3, 7 x 3 = 21, and 10101 is 21. Everything checks out.

We will not be dealing much with binary subtraction, but the process uses the concept of "borrowing" as in base 10 subtraction. The basic rules are:

0 – 0 = 0
1 – 0 = 1
0 – 1 = 1 borrow 1 from the next higher digit
1 – 1 = 0

Example: Subtract 10 from 1101 in binary

$$\begin{array}{r} 1\cancel{1}{}^{1}01 \\ -\quad 10 \\ \hline 10\,11 \end{array}$$

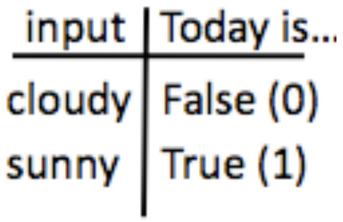
Check? 13 – 2 = 11. All is right with the world.

Boolean Logic and Operations

Lots of things can be considered in terms of binary numbers. For instance, we can interpret the digits 1 and 0 as "yes and no", "true and false", "on and off", or "high and low". As you can see, the binary number system lends itself quite nicely to ideas dealing with logic. The area of mathematics dealing with logic is called Boolean algebra, named after George Boole who developed it in the 19th century. We refer to the digits 1 and 0 as **bits** and the interpretation of the bits ("high", "low", "true", "false", etc.) as **logic states**.

Since Boolean algebra deals with logic, we should be able to take concrete examples from everyday speech and interpret them in terms of bits and logic states. A simple example is the weather. As I'm writing this, it is (very) sunny outside. If someone were to say to me "It's sunny outside", I know that sentence is true because I can see outside. I, therefore, would assign it a value of 1 to signify that it was true. If another person said, “It’s cloudy outside”, which I know to be false, I would assign his statement a value of 0. This simple example regarding the weather is good to keep in mind when dealing with more complex logic statements.

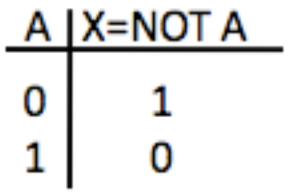
In order to determine in which cases a statement is true or not, we can create a truth table. A **truth table** is a table that contains the values of the inputs (usually labeled “A”, “B”, etc.) in the left-hand columns and the resulting value of the statement (or “output”, usually labeled “X”) in the right-hand column. In our weather example, the inputs are “cloudy” and “sunny” and the statement is “Today is ___(input)___ outside”. Below is the truth table summarizing these results:



input	Today is...
cloudy	False (0)
sunny	True (1)

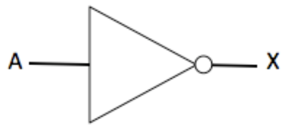
In this example, the inputs “cloudy” (0) and “sunny” (1) are **complementary**, which means the state can only exist in one of these at a time (*i.e.*, it can’t be both sunny and cloudy at the same time).

There are three basic operations that can be performed on a set of inputs—NOT, AND, and OR. (Note that logic operations are written in all capitals.) A device that implements a logical operation is called a **logic gate**. The simplest logic gate, NOT, acts as an “inverter”—it takes one input and outputs its opposite. The truth table for a NOT logic gate is:



A	X=NOT A
0	1
1	0

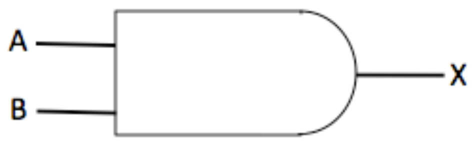
A NOT gate is represented schematically by a triangle with a circle at the output



An AND gate compares two inputs to determine whether or not they are the same. The output of an AND gate is true (1) if and only if both of the inputs (A and B) are true:

A	B	X = A AND B
0	0	0
0	1	0
1	0	0
1	1	1

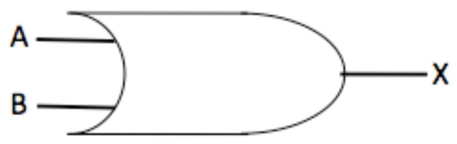
The schematic for an AND gate looks like an extended letter D with two inputs on the left and a single output on the right:



As with an AND gate, an OR gate takes two inputs and compares their values. An OR gate yields an output of 1 as long as at least one of the inputs is 1:

A	B	X = A OR B
0	0	0
0	1	1
1	0	1
1	1	1

An OR logic gate is represented as an arrowhead-like shape:



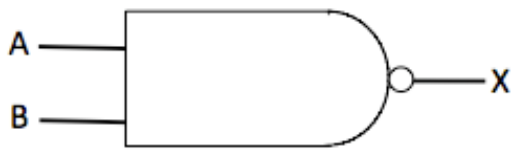
We can make sense of the AND and OR truth tables by thinking about our weather example again. Since I'm writing this in December, it is both sunny and cold outside. Therefore, I know that the sentence "Today is both cloudy and cold" (0 AND 1) is false (0), while "Today is either cloudy or cold" (0 OR 1) is true (1).

By combining the three basic logic gates, we can obtain more complex functions. A NAND gate is essentially an AND gate followed by a NOT gate.

(NAND is short for “NOT AND”.) The output of a NAND gate is the inverse of an AND gate:

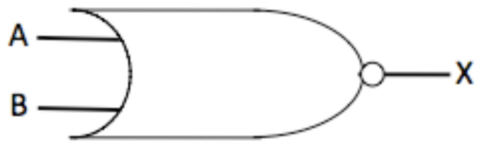
A	B	X = A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

The schematic for a NAND gate is an AND gate with a circle at the output:



Likewise, a NOR gate is an OR gate followed by a NOT gate:

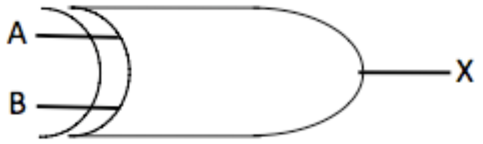
A	B	X = A NOR B
0	0	1
0	1	0
1	0	0
1	1	0



An XOR gate (eXclusive OR) emphasizes the differences between two inputs; XOR is true only if exactly one of the inputs is true, *not both*:

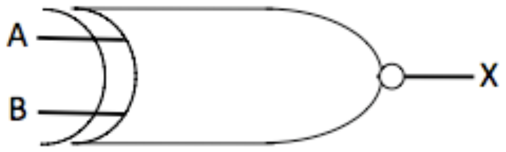
A	B	X = A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

The XOR gate schematic is very similar to an OR gate, but has an extra line near the inputs:



There is also an XNOR gate, which has an inverted output compared to the XOR:

A	B	X = A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1



Logic Identities and Applications

Boolean algebra has many identities and laws, similar to those found in elementary algebra, to simplify logical statements. Logical statements obey an order of operations of sorts. For example, A NAND B is the same as NOT (A AND B). This suggests a simple way of generating complex truth tables—first generate the truth table for A AND B and then invert the output:

A	B	A AND B	NOT(A AND B)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

Two logical statements that have identical outputs are equivalent. If we compare this to the NAND truth table above, we see that A NAND B and NOT (A AND B) are equivalent, as we expected. Another important identity states that two successive NOT gates cancel one another out: NOT (NOT A) = A. This is the same as "double negatives" in the English language (*e.g.*, "I'm not *not* licking toads," Homer J. Simpson).

There is also a distributive property of sorts for logical statements. DeMorgan's theorems are useful for simplifying complex expressions:

NOT (A OR B) = (NOT A) AND (NOT B)
NOT (A AND B) = (NOT A) OR (NOT B)

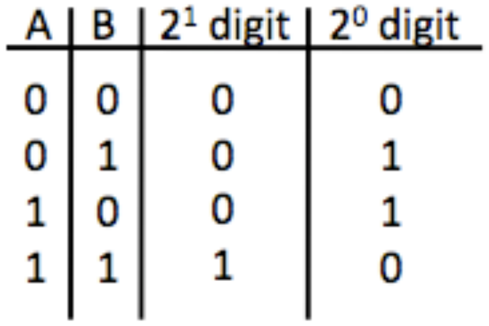
Remember that NOT (A OR B) and NOT (A AND B) are equivalent to A NOR B and A NAND B, respectively. It would be good practice to prove to yourself by creating the necessary truth tables and show that DeMorgan's theorems are indeed equivalent.

An Application of Logic in Digital Circuits: The Half-Adder

Many present-day electronic devices are made up of circuits that perform simple digital logic operations in order to accomplish more complex tasks. One such simple task is adding two binary numbers. Suppose we want to design a logic gate that takes two single-digit binary numbers and adds them together; what would it look like? First of all, we know it will require two inputs (one of each number). We also expect it to output the correct values for binary addition:

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,1 \\ \hline 10 \end{array}$$

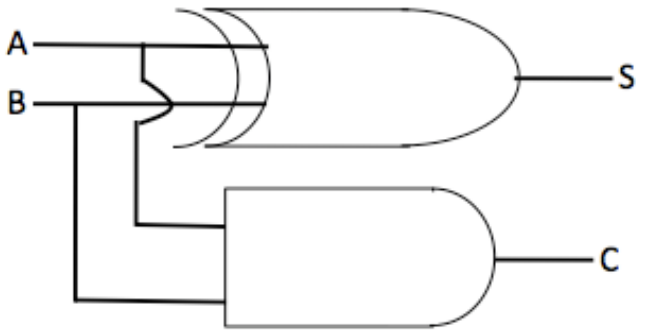
Upon reviewing the results for binary addition, we see we will need *two* outputs—one for the 2^0's digit and one for the 2^1's digit—but all of our logic gates only yield *one* output. Therefore, we'll need to feed our two inputs into *two* separate gates. With all of this information in mind, let's generate a truth table for the addition of two single-digit binary numbers:



A	B	2^1 digit	2^0 digit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

These truth tables look familiar—the 2^0 digit (called the "sum" bit) is equivalent to an XOR gate and the 2^1 digit (called the "carry" bit, as in "carry the one") is equivalent to an AND gate.

The schematic for this device, called a half-adder, looks like:



(The "bump" in the diagram symbolizes that the A input is fed into the AND gate without interfering with the B input of the XOR gate. The outputs S and C refer to the "sum" bit and "carry" bit, respectively.) A half-adder can only be used to add two single-digit binary numbers. In order to perform more complex feats of addition, we would need to use a full-adder, which has *three* inputs: A, B, and a Carry In (C_{in}) bit. We can try adding 11 + 11 in binary to see why three inputs are necessary—once we add the 2^0's digits, we will need to carry the 1 to the 2^1's column where we now have 1 + 1 + 1.

Introduction to Digital Circuits

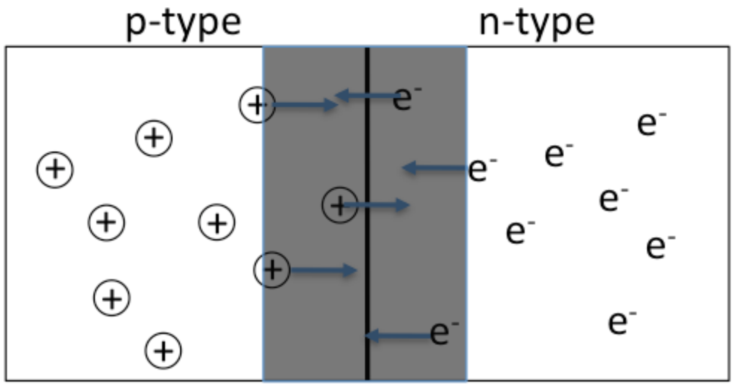
As we saw up above, we can think of an electrical device being "on" or "off" in terms of binary logic. Turning circuits on and off within a device is easily accomplished by introducing a switch into the circuitry. It is advantageous to have an electrical component that acts like a switch, but is *internal* to the circuit. This way I can have the device do the switching on/off for me (rather than a human constantly flipping a switch); also, the device can perform this action much faster than a human could. Ideally, this "switch" would be based on a measurement within the circuit (*e.g.*, potential differences between two points). In this case, we would need to make our switch out of a material that selectively conducts—once a certain voltage measurement reaches a specific threshold, the switch flips. Regular conductors don't act this way (they *always* conduct), but semiconductors do.

Semiconductors

Semiconductors, as the name suggests, are materials with a conductivity between that of conductors and insulators. The conductivity of a semiconductor is dependent upon many properties, such as the material's temperature and its purity. Especially important for our discussion of digital circuits, a semiconductor's conductivity can change in the presence of an external electric field.

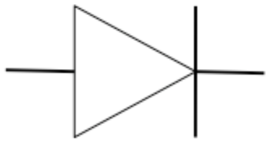
There are a few types of semiconductors. Compounds made up group III and group V elements (*e.g.*, GaAs and InP), as well as II-VI compounds (*e.g.*, ZnTe and ZnO) act as semiconductors. Crystals made solely of elements from group IV of the periodic table (especially silicon and germanium) are called **intrinsic semiconductors**. Impurities can be introduced to adjust the electrical and physical properties of semiconductors by increasing the number of available charge carriers; this process is known as **doping**. For example, if we were to replace one of the silicon atoms in an intrinsic semiconductor with a phosphorus atom (which has five valence electrons), the resulting material has an unpaired electron and results in an **n-type semiconductor** (n for "negative" since it has more electrons). Replacing a silicon atom with an aluminum atom (three valence electrons) yields a **p-type semiconductor** (p for "positive" due to the missing electron). We can think of a missing electron as a positively-charged particle called a **hole**, which will also act as a charge carrier.

A p-type and an n-type semiconductor can be attached to one another to create electrical components and devices. The area where the two semiconductors meet is called a **p-n junction**. At this junction, the charge carriers found just over the border in the two sides (positive holes in the p-type semiconductor, negative electrons in the n-type semiconductor) diffuse into the neighboring semiconductor. When an electron and a hole meet they recombine (*i.e.*, "cancel each other out"); this creates a region of no mobile charge carriers in the neighborhood of the junction known as the **depletion region**. The charge carriers are effectively sequestered from one another because of the depletion region, and no charge can flow:



A diagram of a p-n junction. Electrons and holes diffuse through the boundary and recombine to create the depletion region (grey).

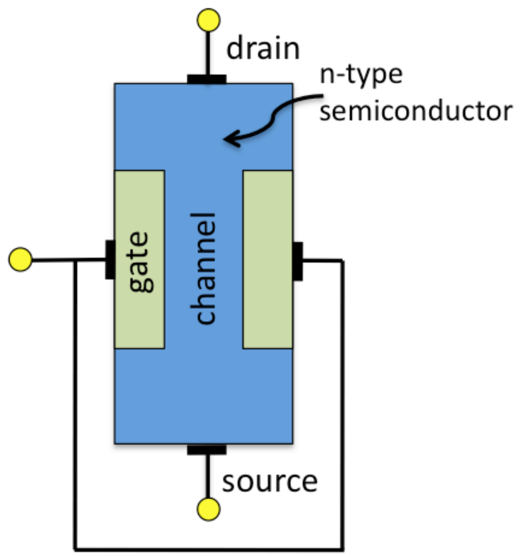
We can create a current by applying an external voltage across the junction. Applying a positive potential to the p-type side and a negative potential to the n-type side will cause the charge carriers to flow toward the junction, shrinking the depletion region until current can flow freely. An external voltage applied in this fashion is called **forward bias**. If we apply a **reverse bias** across the semiconductor, where the n-type side is at a higher potential than the p-type side, the charge carriers will move away from the junction causing the depletion region to grow and, therefore, very little current will flow. A **diode** is an electrical component that has these properties; it only allows current to flow when a forward bias is applied across it. The circuitry symbol for a diode is a triangle (pointing in the direction current can flow) with a vertical line:



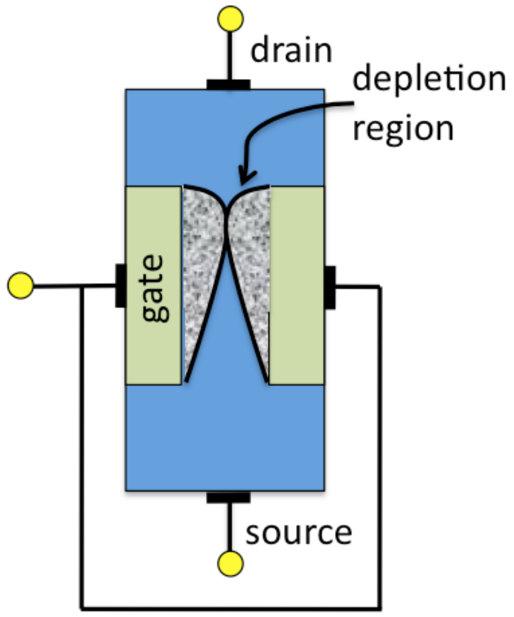
Light-emitting diodes (LEDs) are the most common everyday example of diodes; light is emitted when an electron and a hole annihilate one another.

Field-Effect Transistors and Digital Circuits

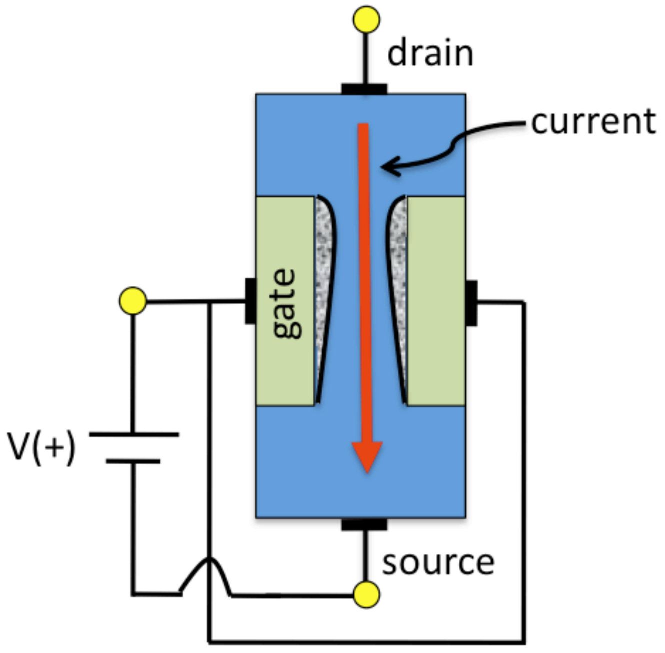
An n-channel field-effect transistor (FET) is a semiconductor-based electronic component that can be used to regulate current flow in a circuit. It has three terminals—the drain, the gate, and the source.



A voltage applied to the gate regulates current flowing through an n-channel FET from the drain to the source, which means the drain terminal should be at a higher potential than the source terminal. Usually, the source is attached to ground. (*NB*: The “n” in “n-channel” refers to negatively-charged particles—electrons—moving from the source to the drain, which translates to a current flow from the drain to the source as stated above.) When there is no external applied voltage to the gate, no current will flow through the transistor due to the presence of a depletion region in the semiconductor.

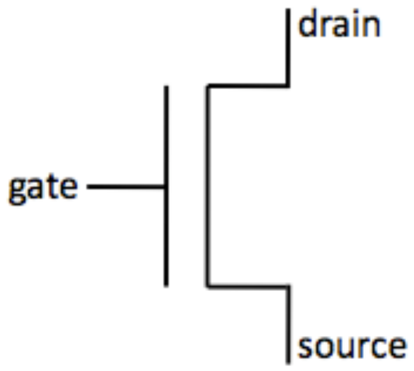


The transistor essentially acts as an open switch in this configuration. As we start to apply an external voltage to the gate and increase its magnitude, the depletion region will shrink until some threshold voltage at which a “channel” forms allowing the charge carriers to flow freely.



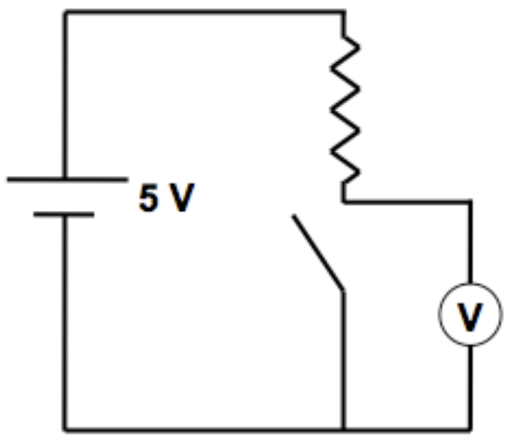
At this point the transistor acts as a closed switch (*i.e.*, a regular wire). Note that there is no electrical connection between the drain and the source; the gate just blocks or allows current through the semiconductor.

The schematic for a transistor is:



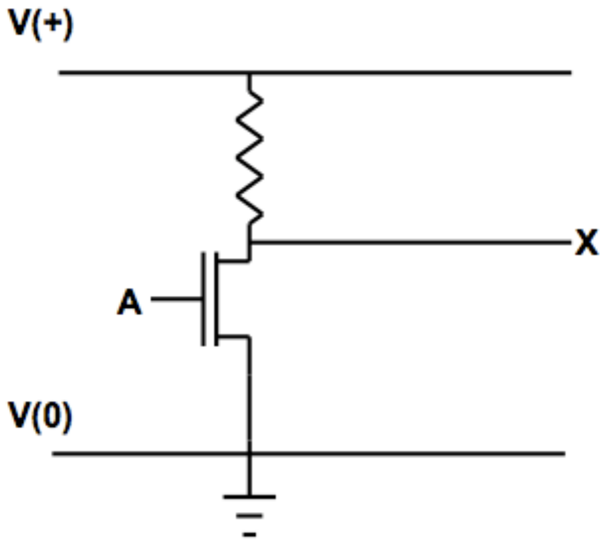
Transistors possess the properties we are looking for regarding our discussion of binary logic—they can act as switches, they are internal to our circuit of interest, and their performance depends on an applied voltage. By combining transistors with other components we can create circuits that have the same properties as the logic gates we saw earlier.

Before looking at a digital circuit, let’s take the following circuit:



As drawn, the switch is open and no current will flow through the resistor. Since there is no current flowing through the resistor, there will be no voltage drop across the resistor. Therefore, the voltmeter will read 5 V. When the switch is closed, current is free to flow through the resistor, which causes a potential drop across the resistor, and the voltmeter will read 0 V.

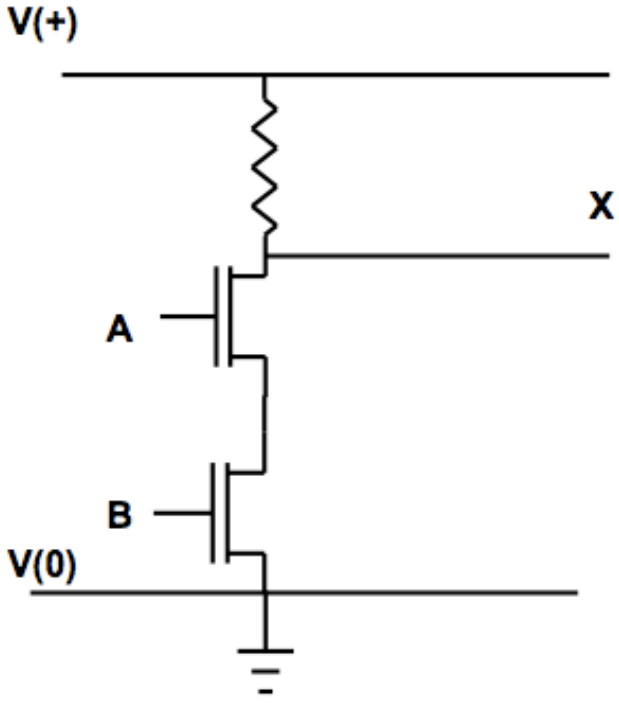
Now let's replace the switch with a transistor:



Although drawn a little differently, this circuit is analogous to the previous one. Since we are using digital circuits to perform binary logic, we are not interested in the exact value of voltages in our digital circuits—only if they are high (1) or low (0). With this in mind, we use the symbol V(+) to designate high potential (usually +5 V) and V(0) designates low potential (usually ground). We can interpret the external voltage applied to the gate as an input (A). If the input is 0 (below the threshold voltage), then the transistor acts as an open switch; if the input is 1 (above the threshold voltage), then it acts as a wire with no resistance. The potential difference between the output (X), located at the drain terminal of the transistor, and ground is the main voltage measurement we are interested in. This is the same measurement as the voltmeter in our previous circuit. It is important to keep in mind that the output itself (X) does not draw any current. This ensures that there will *only* be a voltage drop across the resistor when the input of the transistor is 1. (*NB*: The resistor is known as a "pull-up resistor" and is used to enforce a reliable, known voltage drop.)

In order for a digital circuit to be used as a logic gate, it should have the same behavior at the gate. This digital circuit is an implementation of the simplest logic gate, X = NOT A. An input of 0 means the transistor acts as an open switch. Since no current is flowing through the resistor, V(X) = V(+). The output of the gate corresponds to the potential difference between X and ground, which in this case is V(+) or 1 ("high"). Using the same method it is easy to see that an input of 1 will yield an output of 0. A device that outputs the *opposite value* as its input is exactly what we would expect for a NOT logic gate.

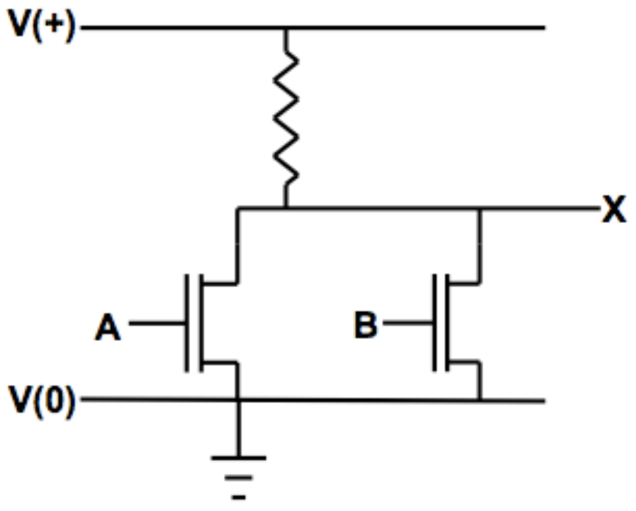
Digital circuits that implement other logic gates will require more than one transistor. Since the gate terminal of a transistor corresponds to one input, we will need a transistor for *each* input of our logic gate. The following digital circuit is equivalent to X = A NAND B:



The first thing we notice is that there are two transistors in this circuit—one for input A, another for input B—and that they are wired in series. The output X is still measured with respect to the drain leg of the transistor immediately following the pull-up resistor. No current will flow through the resistor unless *both* A *and* B are equal to 1. This means that voltage measured at X will be V(+) (high, 1), except for A = B = 1 when the measured voltage will be V(0) (low, 0). Recall the truth table for a NAND gate:

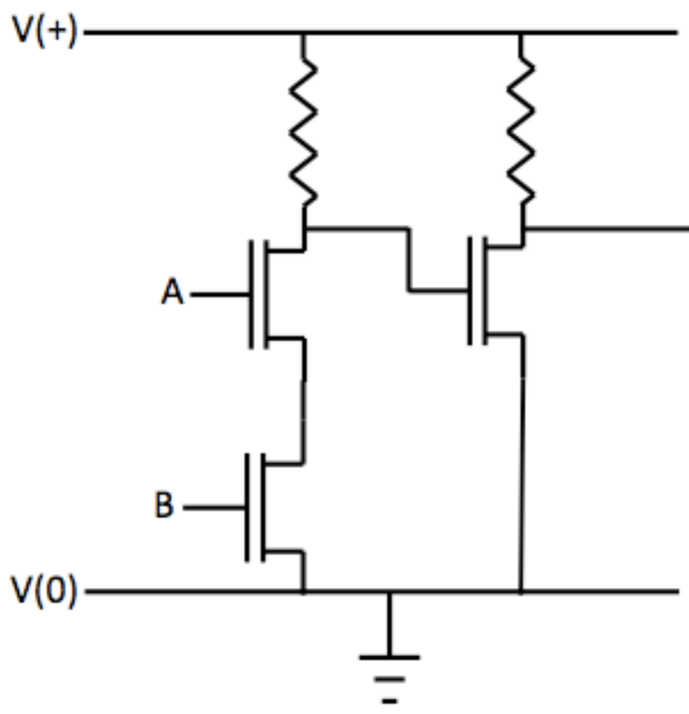
A	B	X = A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

The statement A NAND B is true *except* when both A and B are true. The circuit corresponding to a NOR gate also uses two transistors:



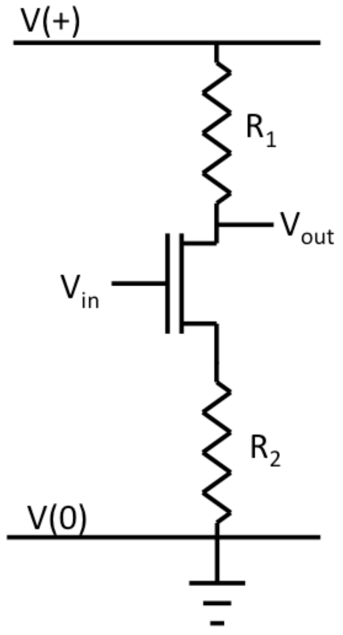
Unlike the NAND circuit *either* A *or* B needs to equal 1 for current to flow through the resistor, which is the expected behavior for a NOR gate.

Some logic gates are created by combining the basic digital circuits we've already seen. Having the output of one gate become the input of a subsequent gate is one way of achieving this. The statement X = (A AND B), for example, is equivalent to NOT (A NAND B). To build this AND gate, we would first build the circuit for A NAND B. The output of the NAND circuit would become the input for a NOT gate:



It is worth proving to yourself that this circuit does indeed correspond to an AND gate given the various combinations of inputs for A and B.

In certain configurations, a transistor in a circuit can act as an amplifier, where the output voltage or current of the transistor is larger than its input value. The ratio of the output value to the input value is known as the gain. For example the following circuit, known as a common-source amplifier, acts as a voltage amplifier:

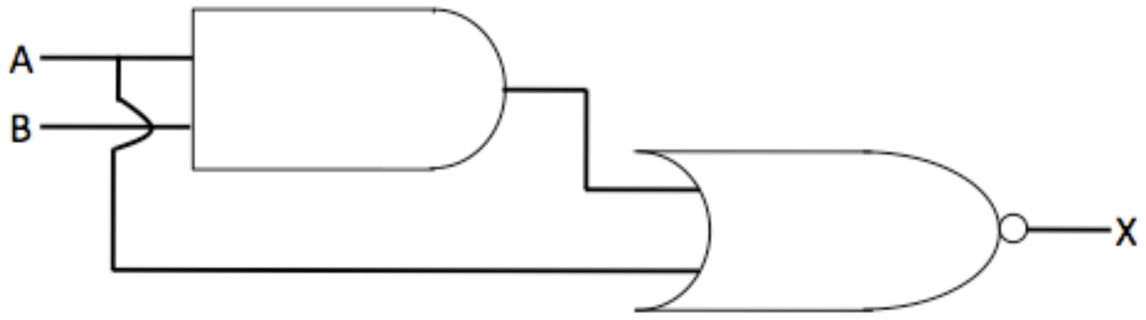


The voltage gain produced by this circuit is dependent upon the relative values of the two resistors, R_1 and R_2.

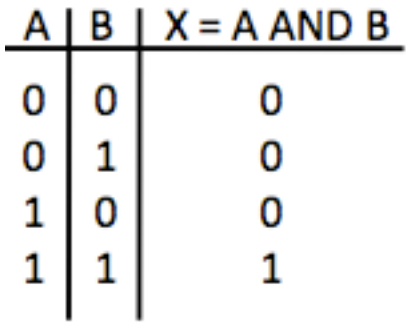
Problems Involving Digital Circuits

There are three main elements when dealing with digital circuits—gate diagrams, circuit diagrams, and truth tables. Most problems will involve generating one of these elements given some background knowledge. The half-adder example from above is an example of generating a gate diagram from a truth table. In the lab you will make a half-adder by designing and actually wiring up a digital circuit that corresponds to your gate diagram. In the following sections, we'll work through a couple of examples to get a feel for the types of questions you could be asked regarding digital circuits.

First, let's generate a truth table for a given gate diagram:



The first step in approaching this type of problem is to identify the logic gates in play and their inputs. The first gate is an AND gate with two inputs, A and B. The second gate is a NOR gate with two inputs, A and the *output* from the AND gate. This gives us a place to start—we know that we need to generate the truth table for the first gate before dealing with the second gate. We know the truth table for an AND gate already:



A	B	X = A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Now we need to take the output from this gate as one of the inputs for our NOR gate and determine the truth table:

A	A AND B	X = (A AND B) NOR A
0	0	1
0	0	1
1	0	0
1	1	0

Another method utilizes one large truth table containing each intermediate step:

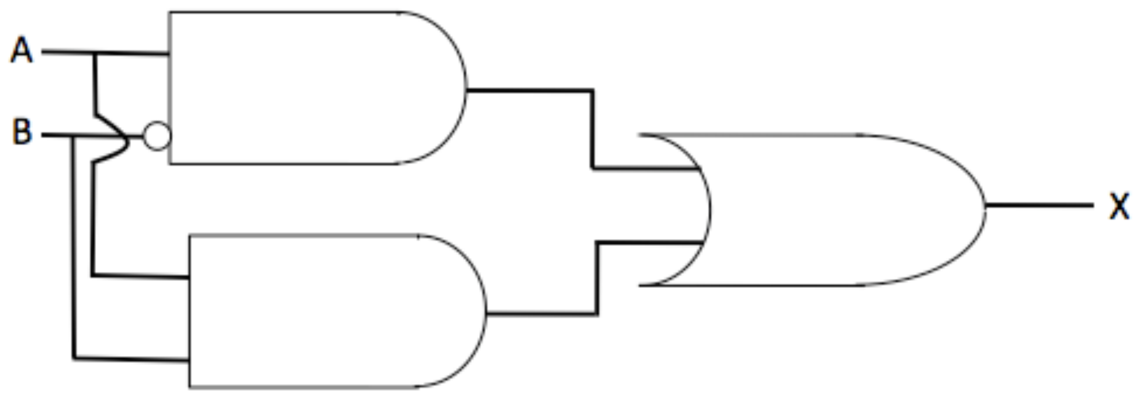
A	B	A AND B	(A AND B) OR A	X = (A AND B) NOR A
0	0	0	0	1
0	1	0	0	1
1	0	0	1	0
1	1	1	1	0

After determining the “A AND B” column from the first gate, we can easily generate the “(A AND B) OR A” column by covering up the “B” column with our finger and looking at the “A” and “A AND B” columns as inputs. We are interested in the final output of the NOR gate, which is obtained by negating the entries in the “(A AND B) OR A” column.

Going from a truth table to a gate diagram is a little trickier. We are interested only in each instance when the output is true (1) for a given set of inputs when interpreting the truth table. Take this truth table, for example:

A	B	X
0	0	0
0	1	0
1	0	1
1	1	1

This gate is only true for the last two rows of the truth table. The third row says that X is true if A is true and B is false. Written in terms of logic gates, this line would read X = A AND (NOT B). The gate is also true when A is true and B is true, or X = A AND B. Since either of these sets of inputs are valid, a complete solution for the gate is X = (A AND (NOT B)) OR (A AND B). The OR gate implies that there is more than one set of inputs that satisfies the truth table. Although messy, the gate diagram for this solution is:



A much simpler solution can be obtained if we think about what the entries in the truth table mean. Notice that the output X is 1 whenever A is 1, *regardless of what B is*! Therefore, the simplest solution to this truth table is simply X = A. A brute force approach will work every time, but thinking about the truth table first may lead to a much simpler (and intuitive) answer.