2. Digital Circuits

7.BINARY ARITHMETIC

Binary addition

Adding binary numbers is a very simple task, and very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left. Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10
1 + 1 + 1 = 11
```

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples:

```
.                          11  1 <--- Carry bits ----->  11
.     1001101            1001001                     1000111
.   + 0010010          + 0011001                   + 0010110
.   ---------          ---------                   ---------
.     1011111            1100010                     1011101
```

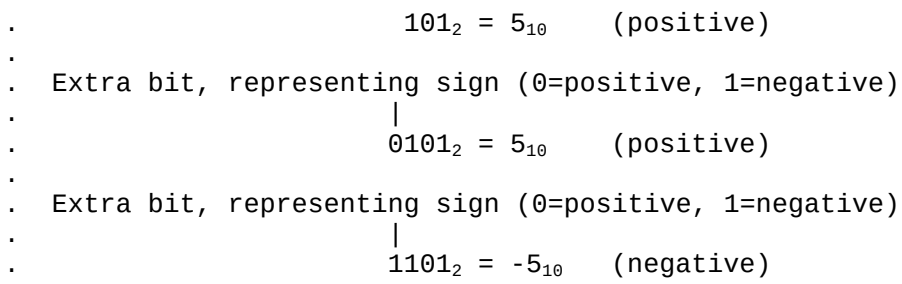
The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

As we'll see later, there are ways that electronic circuits can be built to perform this very task of addition, by representing each bit of each binary number as a voltage signal (either "high," for a 1; or "low" for a 0). This is the very foundation of all the arithmetic which modern digital computers perform.

Negative binary numbers

With addition being easily accomplished, we can perform the operation of subtraction with the same technique simply by making one of the numbers negative. For example, the subtraction problem of 7 - 5 is essentially the same as the addition problem 7 + (-5). Since we already know how to represent positive numbers in binary, all we need to know now is how to represent their negative counterparts and we'll be able to subtract.

Usually we represent a negative decimal number by placing a minus sign directly to the left of the most significant digit, just as in the example above, with -5. However, the whole purpose of using binary notation is for constructing on/off circuits that can represent bit values in terms of voltage (2 alternative values: either "high" or "low"). In this context, we don't have the luxury of a third symbol such as a "minus" sign, since these circuits can only be on or off (two possible states). One solution is to reserve a bit (circuit) that does nothing but represent the mathematical sign:



As you can see, we have to be careful when we start using bits for any purpose other than standard place-weighted values. Otherwise, 1101_2 could be misinterpreted as the number thirteen when in fact we mean to represent negative five. To keep things straight here, we must first decide how many bits are going to be needed to represent the largest numbers we'll be dealing with, and then be sure not to exceed that bit field length in our arithmetic operations. For the above example, I've limited myself to the representation of numbers from negative seven (1111_2) to positive seven (0111_2), and no more, by making the fourth bit the "sign" bit. Only by first establishing these limits can I avoid confusion of a negative number with a larger, positive number.

Representing negative five as 1101_2 is an example of the *sign-magnitude* system of negative binary numeration. By using the leftmost bit as a sign indicator and not a place-weighted value, I am sacrificing the "pure" form of binary notation for something that gives me a practical advantage: the representation of negative numbers. The leftmost bit is read as the sign, either positive or negative, and the remaining bits are interpreted according to the standard binary notation: left to right, place weights in multiples of two.

As simple as the sign-magnitude approach is, it is not very practical for arithmetic purposes. For instance, how do I add a negative five (1101_2) to any other number, using the standard technique for binary addition? I'd have to invent a new way of doing addition in order for it to work, and if I do that, I might as well just do the job with longhand subtraction; there's no arithmetical advantage to using negative numbers to perform subtraction through addition if we have to do it with sign-magnitude numeration, and that was our goal!

There's another method for representing negative numbers which works with our familiar technique of longhand addition, and also happens to make more sense from a place-weighted numeration point of view, called *complementation*. With this strategy, we assign the leftmost bit to serve a special purpose, just as we did with the sign-magnitude approach, defining our number limits just as before. However, this time, the leftmost bit is more than just a sign bit; rather, it possesses a negative place-weight value. For example, a value of negative five would be represented as such:

```
Extra bit, place weight = negative eight
.                    |
.                    1011₂ = 5₁₀   (negative)
.
.          (1 x -8₁₀)  +  (0 x 4₁₀)  +  (1 x 2₁₀)  +  (1 x 1₁₀)  =  -5₁₀
```

With the right three bits being able to represent a magnitude from zero through seven, and the leftmost bit representing either zero or negative eight, we can successfully represent any integer number from negative seven ($1001_2 = -8_{10} + 1_{10} = -1_{10}$) to positive seven ($0111_2 = 0_{10} + 7_{10} = 7_{10}$).

Representing positive numbers in this scheme (with the fourth bit designated as the negative weight) is no different from that of ordinary binary notation. However, representing negative numbers is not quite as straightforward:

```
zero             0000
positive one     0001          negative one     1111
positive two     0010          negative two     1110
positive three   0011          negative three   1101
positive four    0100          negative four    1100
positive five    0101          negative five    1011
positive six     0110          negative six     1010
positive seven   0111          negative seven   1001
.                              negative eight   1000
```

Note that the negative binary numbers in the right column, being the sum of the right three bits' total plus the negative eight of the leftmost bit, don't "count" in the same progression as the positive binary numbers in the left column. Rather, the right three bits have to be set at the proper value to equal the desired (negative) total when summed with the negative eight place value of the leftmost bit.

Those right three bits are referred to as the *two's complement* of the corresponding positive number. Consider the following comparison:

```
positive number       two's complement
---------------       ----------------
001                    111
010                    110
011                    101
100                    100
101                    011
110                    010
111                    001
```

In this case, with the negative weight bit being the fourth bit (place value of negative eight), the two's complement for any positive number will be whatever value is needed to add to

negative eight to make that positive value's negative equivalent. Thankfully, there's an easy way to figure out the two's complement for any binary number: simply invert all the bits of that number, changing all 1's to 0's and vice versa (to arrive at what is called the *one's complement*) and then add one! For example, to obtain the two's complement of five (101_2), we would first invert all the bits to obtain 010_2 (the "one's complement"), then add one to obtain 011_2, or -5_{10} in three-bit, two's complement form.

Interestingly enough, generating the two's complement of a binary number works the same if you manipulate *all* the bits, including the leftmost (sign) bit at the same time as the magnitude bits. Let's try this with the former example, converting a positive five to a negative five, but performing the complementation process on all four bits. We must be sure to include the 0 (positive) sign bit on the original number, five (0101_2). First, inverting all bits to obtain the one's complement: 1010_2. Then, adding one, we obtain the final answer: 1011_2, or -5_{10} expressed in four-bit, two's complement form.

It is critically important to remember that the place of the negative-weight bit must be already determined before any two's complement conversions can be done. If our binary numeration field were such that the eighth bit was designated as the negative-weight bit (10000000_2), we'd have to determine the two's complement based on all seven of the other bits. Here, the two's complement of five (0000101_2) would be 1111011_2. A positive five in this system would be represented as 00000101_2, and a negative five as 11111011_2.

Subtraction

We can subtract one binary number from another by using the standard techniques adapted for decimal numbers (subtraction of each bit pair, right to left, "borrowing" as needed from bits to the left). However, if we can leverage the already familiar (and easier) technique of binary addition to subtract, that would be better. As we just learned, we can represent negative binary numbers by using the "two's complement" method and a negative place-weight bit. Here, we'll use those negative binary numbers to subtract through addition. Here's a sample problem:

Subtraction: $7_{10} - 5_{10}$ Addition equivalent: $7_{10} + (-5_{10})$

If all we need to do is represent seven and negative five in binary (two's complemented) form, all we need is three bits plus the negative-weight bit:

positive seven = 0111_2
negative five = 1011_2

Now, let's add them together:

```
.                    1111  <--- Carry bits
.                     0111
.                   + 1011
.                   ------
.                    10010
.                    |
.             Discard extra bit
.
.            Answer = 0010₂
```

Since we've already defined our number bit field as three bits plus the negative-weight bit, the fifth bit in the answer (1) will be discarded to give us a result of 0010_2, or positive two, which is the correct answer.

Another way to understand why we discard that extra bit is to remember that the leftmost bit of the lower number possesses a negative weight, in this case equal to negative eight. When we add these two binary numbers together, what we're actually doing with the MSBs is subtracting the lower number's MSB from the upper number's MSB. In subtraction, one never "carries" a digit or bit on to the next left place-weight.

Let's try another example, this time with larger numbers. If we want to add -25_{10} to 18_{10}, we must first decide how large our binary bit field must be. To represent the largest (absolute value) number in our problem, which is twenty-five, we need at least five bits, plus a sixth bit for the negative-weight bit. Let's start by representing positive twenty-five, then finding the two's complement and putting it all together into one numeration:

$+25_{10} = 011001_2$ (showing all six bits)
One's complement of $11001_2 = 100110_2$
One's complement + 1 = two's complement = 100111_2
$-25_{10} = 100111_2$

Essentially, we're representing negative twenty-five by using the negative-weight (sixth) bit with a value of negative thirty-two, plus positive seven (binary 111_2).

Now, let's represent positive eighteen in binary form, showing all six bits:

```
.              18₁₀  = 010010₂
.
.       Now, let's add them together and see what we get:
.
.                      11   <--- Carry bits
.                   100111
.                 + 010010
.                 --------
.                   111001
```

Since there were no "extra" bits on the left, there are no bits to discard. The leftmost bit on the answer is a 1, which means that the answer is negative, in two's complement form, as it should be. Converting the answer to decimal form by summing all the bits times their respective weight values, we get:

$$(1 \times -32_{10}) + (1 \times 16_{10}) + (1 \times 8_{10}) + (1 \times 1_{10}) = -7_{10}$$

Indeed -7_{10} is the proper sum of -25_{10} and 18_{10}.

Overflow

One caveat with signed binary numbers is that of *overflow*, where the answer to an addition or subtraction problem exceeds the magnitude which can be represented with the alloted number of bits. Remember that the place of the sign bit is fixed from the beginning of the problem. With the last example problem, we used five binary bits to represent the magnitude of the number, and the left-most (sixth) bit as the negative-weight, or sign, bit. With five bits to represent magnitude, we have a representation range of 2^5, or thirty-two integer steps from 0 to maximum. This means that we can represent a number as high as $+31_{10}$ (011111_2), or as low as -32_{10} (100000_2). If we set up an addition problem with two binary numbers, the sixth bit used for sign, and the result either exceeds $+31_{10}$ or is less than -32_{10}, our answer will be incorrect. Let's try adding 17_{10} and 19_{10} to see how this overflow condition works for excessive positive numbers:

```
.       17₁₀  = 10001₂            19₁₀  = 10011₂
.
.                              1  11  <--- Carry bits
.    (Showing sign bits)       010001
.                            + 010011
.                              --------
.                              100100
```

The answer (100100_2), interpreted with the sixth bit as the -32_{10} place, is actually equal to -28_{10}, not $+36_{10}$ as we should get with $+17_{10}$ and $+19_{10}$ added together! Obviously, this is not correct. What went wrong? The answer lies in the restrictions of the six-bit number field within which we're working Since the magnitude of the true and proper sum (36_{10}) exceeds the allowable limit for our designated bit field, we have an *overflow error*. Simply put, six places doesn't give enough bits to represent the correct sum, so whatever figure we obtain using the strategy of discarding the left-most "carry" bit will be incorrect.

A similar error will occur if we add two negative numbers together to produce a sum that is too low for our six-bit binary field. Let's try adding -17_{10} and -19_{10} together to see how this works (or doesn't work, as the case may be!):

```
.      -17₁₀  = 101111₂           -19₁₀  = 101101₂
.
.                            1 1111  <--- Carry bits
.    (Showing sign bits)      101111
.                           + 101101
.                           --------
.                            1011100
.                            |
.                   Discard extra bit
.
FINAL ANSWER:   011100₂   = +28₁₀
```

The (incorrect) answer is a *positive* twenty-eight. The fact that the real sum of negative seventeen and negative nineteen was too low to be properly represented with a five bit magnitude field and a sixth sign bit is the root cause of this difficulty.

Let's try these two problems again, except this time using the seventh bit for a sign bit, and allowing the use of 6 bits for representing the magnitude:

```
.          17₁₀ + 19₁₀                      (-17₁₀) + (-19₁₀)
.
.           1  11                             11 1111
.          0010001                            1101111
.        + 0010011                          + 1101101
.        ---------                          ---------
.          0100100₂                          11011100₂
.                                             |
.                                    Discard extra bit
.
. ANSWERS:  0100100₂ = +36₁₀
.           1011100₂ = -36₁₀
```

By using bit fields sufficiently large to handle the magnitude of the sums, we arrive at the correct answers.

In these sample problems we've been able to detect overflow errors by performing the addition problems in decimal form and comparing the results with the binary answers. For example, when adding $+17_{10}$ and $+19_{10}$ together, we knew that the answer was *supposed* to be $+36_{10}$, so when the binary sum checked out to be -28_{10}, we knew that something had to be wrong. Although this is a valid way of detecting overflow, it is not very efficient. After all, the whole idea of complementation is to be able to reliably add binary numbers together and not have to double-check the result by adding the same numbers together in decimal form! This is especially true for the purpose of building electronic circuits to add binary quantities together: the circuit has to be able to check itself for overflow without the supervision of a human being who already knows what the correct answer is.

What we need is a simple error-detection method that doesn't require any additional arithmetic. Perhaps the most elegant solution is to check for the *sign* of the sum and compare it against the signs of the numbers added. Obviously, two positive numbers added together should give a positive result, and two negative numbers added together should give a negative

result. Notice that whenever we had a condition of overflow in the example problems, the sign of the sum was always *opposite* of the two added numbers: $+17_{10}$ plus $+19_{10}$ giving -28_{10}, or -17_{10} plus -19_{10} giving $+28_{10}$. By checking the signs alone we are able to tell that something is wrong.

But what about cases where a positive number is added to a negative number? What sign should the sum be in order to be correct. Or, more precisely, what sign of sum would necessarily indicate an overflow error? The answer to this is equally elegant: there will *never* be an overflow error when two numbers of opposite signs are added together! The reason for this is apparent when the nature of overflow is considered. Overflow occurs when the magnitude of a number exceeds the range allowed by the size of the bit field. The sum of two identically-signed numbers may very well exceed the range of the bit field of those two numbers, and so in this case overflow is a possibility. However, if a positive number is added to a negative number, the sum will always be closer to zero than either of the two added numbers: its magnitude *must* be less than the magnitude of either original number, and so overflow is impossible.

Fortunately, this technique of overflow detection is easily implemented in electronic circuitry, and it is a standard feature in digital adder circuits: a subject for a later chapter.

Bit groupings

The singular reason for learning and using the binary numeration system in electronics is to understand how to design, build, and troubleshoot circuits that represent and process numerical quantities in digital form. Since the bivalent (two-valued) system of binary bit numeration lends itself so easily to representation by "on" and "off" transistor states (saturation and cutoff, respectively), it makes sense to design and build circuits leveraging this principle to perform binary calculations.

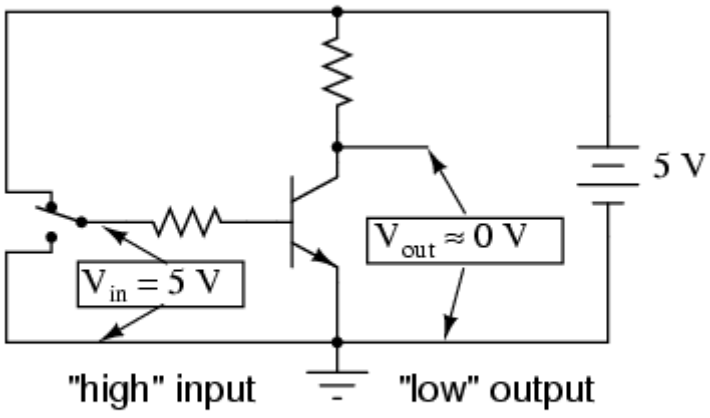
8.LOGIC GATES

Digital signals and gates

While the binary numeration system is an interesting mathematical abstraction, we haven't yet seen its practical application to electronics. This chapter is devoted to just that: practically applying the concept of binary bits to circuits. What makes binary numeration so important to the application of digital electronics is the ease in which bits may be represented in physical terms. Because a binary bit can only have one of two different values, either 0 or 1, any physical medium capable of switching between two saturated states may be used to represent a bit. Consequently, any physical system capable of representing binary bits is able to represent numerical quantities, and potentially has the ability to manipulate those numbers. This is the basic concept underlying digital computing.

Electronic circuits are physical systems that lend themselves well to the representation of binary numbers. Transistors, when operated at their bias limits, may be in one of two different states: either cutoff (no controlled current) or saturation (maximum controlled current). If a transistor circuit is designed to maximize the probability of falling into either one of these states (and not operating in the linear, or *active*, mode), it can serve as a physical representation of a binary bit. A voltage signal measured at the output of such a circuit may also serve as a representation of a single bit, a low voltage representing a binary "0" and a (relatively) high voltage representing a binary "1." Note the following transistor circuit:

Transistor in saturation



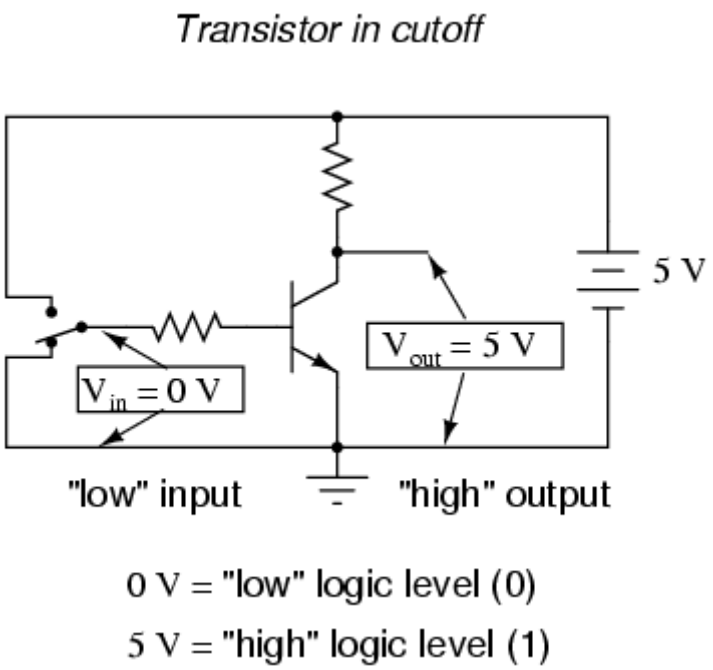
0 V = "low" logic level (0)

5 V = "high" logic level (1)

In this circuit, the transistor is in a state of saturation by virtue of the applied input voltage (5 volts) through the two-position switch. Because it's saturated, the transistor drops very little voltage between collector and emitter, resulting in an output voltage of (practically) 0 volts. If we were using this circuit to represent binary bits, we would say that the input signal is a binary "1" and that the output signal is a binary "0." Any voltage close to full supply voltage (measured in reference to ground, of course) is considered a "1" and a lack of voltage is considered a "0." Alternative terms for these voltage levels are *high* (same as a binary "1")

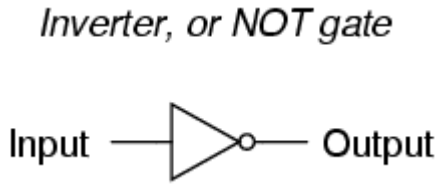
and *low* (same as a binary "0"). A general term for the representation of a binary bit by a circuit voltage is *logic level*.

Moving the switch to the other position, we apply a binary "0" to the input and receive a binary "1" at the output:

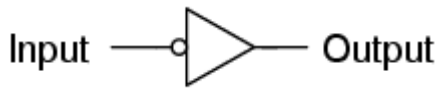


What we've created here with a single transistor is a circuit generally known as a *logic gate*, or simply *gate*. A gate is a special type of amplifier circuit designed to accept and generate voltage signals corresponding to binary 1's and 0's. As such, gates are not intended to be used for amplifying analog signals (voltage signals *between* 0 and full voltage). Used together, multiple gates may be applied to the task of binary number storage (memory circuits) or manipulation (computing circuits), each gate's output representing one bit of a multi-bit binary number. Just how this is done is a subject for a later chapter. Right now it is important to focus on the operation of individual gates.

The gate shown here with the single transistor is known as an *inverter*, or NOT gate, because it outputs the exact opposite digital signal as what is input. For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors. The following is the symbol for an inverter:

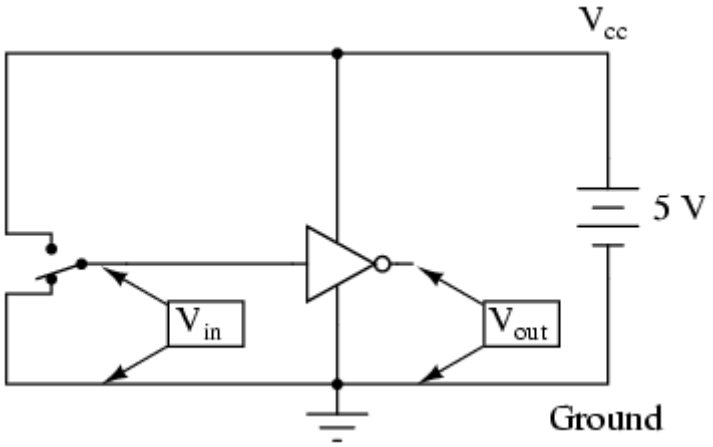


An alternative symbol for an inverter is shown here:

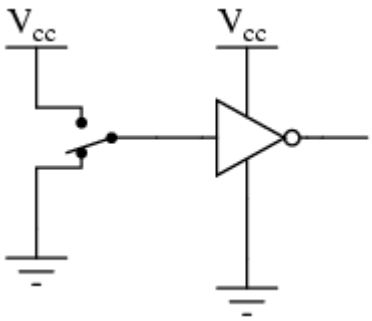


Notice the triangular shape of the gate symbol, much like that of an operational amplifier. As was stated before, gate circuits actually are amplifiers. The small circle, or "bubble" shown on either the input or output terminal is standard for representing the inversion function. As you might suspect, if we were to remove the bubble from the gate symbol, leaving only a triangle, the resulting symbol would no longer indicate inversion, but merely direct amplification. Such a symbol and such a gate actually do exist, and it is called a *buffer*, the subject of the next section.

Like an operational amplifier symbol, input and output connections are shown as single wires, the implied reference point for each voltage signal being "ground." In digital gate circuits, ground is almost always the negative connection of a single voltage source (power supply). Dual, or "split," power supplies are seldom used in gate circuitry. Because gate circuits are amplifiers, they require a source of power to operate. Like operational amplifiers, the power supply connections for digital gates are often omitted from the symbol for simplicity's sake. If we were to show *all* the necessary connections needed for operating this gate, the schematic would look something like this:



Power supply conductors are rarely shown in gate circuit schematics, even if the power supply connections at each gate are. Minimizing lines in our schematic, we get this:

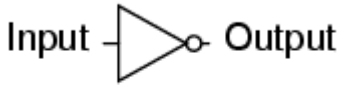


"V_{cc}" stands for the constant voltage supplied to the collector of a bipolar junction transistor circuit, in reference to ground. Those points in a gate circuit marked by the label "V_{cc}" are all connected to the same point, and that point is the positive terminal of a DC voltage source, usually 5 volts.

As we will see in other sections of this chapter, there are quite a few different types of logic gates, most of which have multiple input terminals for accepting more than one signal. The output of any gate is dependent on the state of its input(s) and its logical function.

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." For the inverter, or NOT, circuit just illustrated, the truth table is very simple indeed:

NOT gate truth table



Input	Output
0	1
1	0

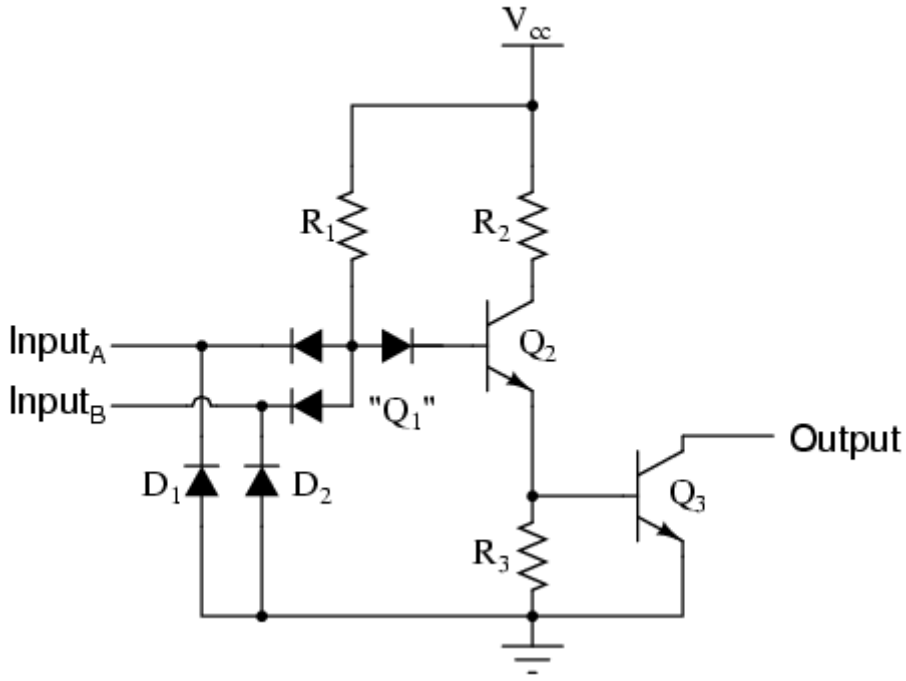
Truth tables for more complex gates are, of course, larger than the one shown for the NOT gate. A gate's truth table must have as many rows as there are possibilities for unique input combinations. For a single-input gate like the NOT gate, there are only two possibilities, 0 and 1. For a two input gate, there are *four* possibilities (00, 01, 10, and 11), and thus four rows to the corresponding truth table. For a three-input gate, there are *eight* possibilities (000, 001, 010, 011, 100, 101, 110, and 111), and thus a truth table with eight rows are needed. The mathematically inclined will realize that the number of truth table rows needed for a gate is equal to 2 raised to the power of the number of input terminals.

- **REVIEW:**
- In digital circuits, binary bit values of 0 and 1 are represented by voltage signals measured in reference to a common circuit point called *ground*. An absence of voltage represents a binary "0" and the presence of full DC supply voltage represents a binary "1."
- A *logic gate*, or simply *gate*, is a special form of amplifier circuit designed to input and output *logic level* voltages (voltages intended to represent binary bits). Gate circuits are most commonly represented in a schematic by their own unique symbols rather than by their constituent transistors and resistors.
- Just as with operational amplifiers, the power supply connections to gates are often omitted in schematic diagrams for the sake of simplicity.
- A *truth table* is a standard way of representing the input/output relationships of a gate circuit, listing all the possible input logic level combinations with their respective output logic levels.

TTL NAND and AND gates

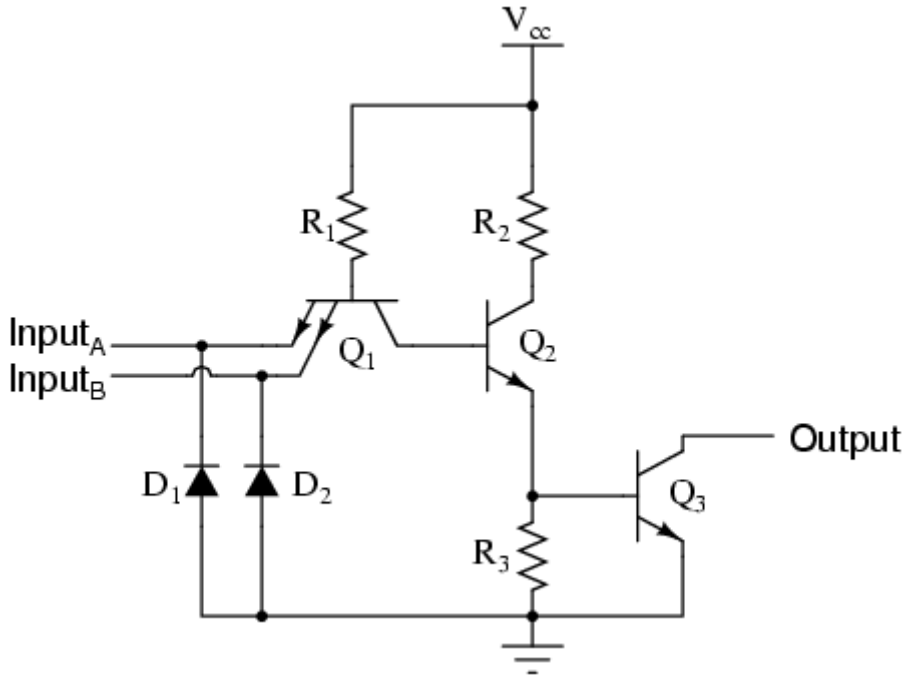
Suppose we altered our basic open-collector inverter circuit, adding a second input terminal just like the first:

A two-input inverter circuit



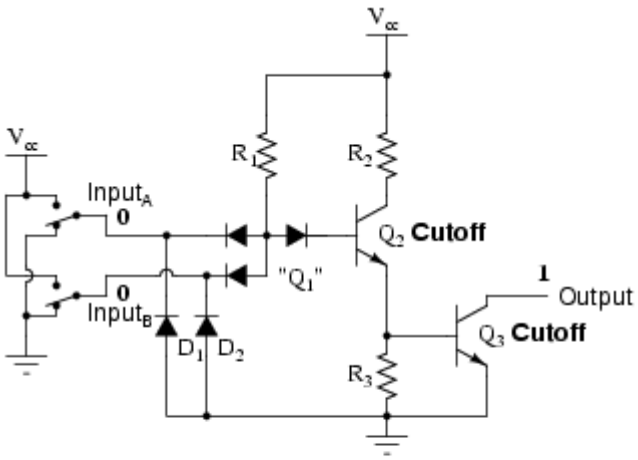
This schematic illustrates a real circuit, but it isn't called a "two-input inverter." Through analysis we will discover what this circuit's logic function is and correspondingly what it should be designated as.

Just as in the case of the inverter and buffer, the "steering" diode cluster marked "Q_1" is actually formed like a transistor, even though it isn't used in any amplifying capacity. Unfortunately, a simple NPN transistor structure is inadequate to simulate the *three* PN junctions necessary in this diode network, so a different transistor (and symbol) is needed. This transistor has one collector, one base, and *two* emitters, and in the circuit it looks like this:



In the single-input (inverter) circuit, grounding the input resulted in an output that assumed the "high" (1) state. In the case of the open-collector output configuration, this "high" state was simply "floating." Allowing the input to float (or be connected to V_{cc}) resulted in the output becoming grounded, which is the "low" or 0 state. Thus, a 1 in resulted in a 0 out, and vice versa.

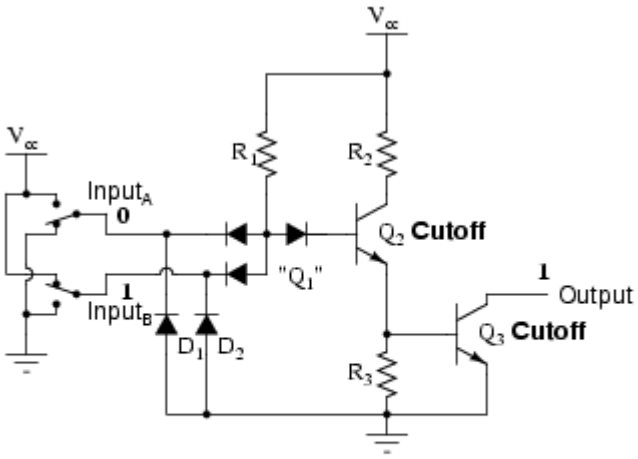
Since this circuit bears so much resemblance to the simple inverter circuit, the only difference being a second input terminal connected in the same way to the base of transistor Q_2, we can say that each of the inputs will have the same effect on the output. Namely, if either of the inputs are grounded, transistor Q_2 will be forced into a condition of cutoff, thus turning Q_3 off and floating the output (output goes "high"). The following series of illustrations shows this for three input states (00, 01, and 10):



$Input_A$ = **0**

$Input_B$ = **0**

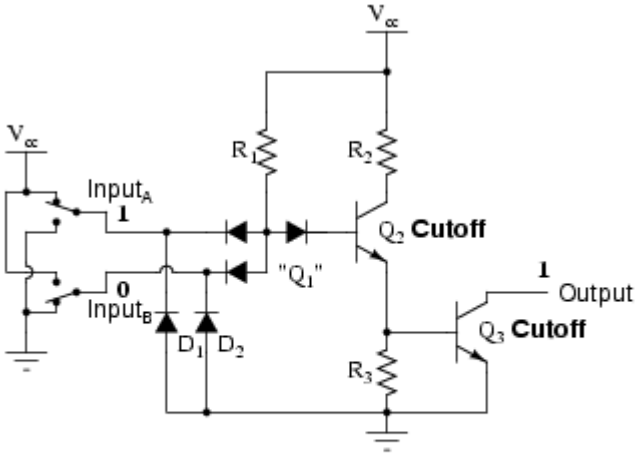
Output = **1**



Input$_A$ = **0**

Input$_B$ = **1**

Output = **1**

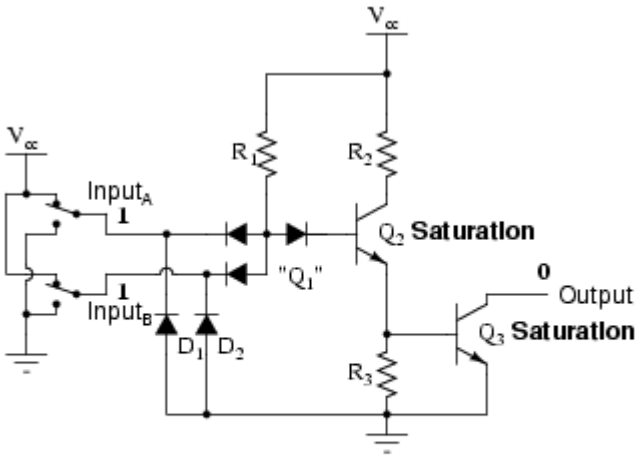


Input$_A$ = **1**

Input$_B$ = **0**

Output = **1**

In any case where there is a grounded ("low") input, the output is guaranteed to be floating ("high"). Conversely, the only time the output will ever go "low" is if transistor Q_3 turns on, which means transistor Q_2 must be turned on (saturated), which means neither input can be diverting R_1 current away from the base of Q_2. The only condition that will satisfy this requirement is when both inputs are "high" (1):



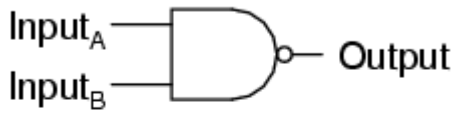
$Input_A$ = **1**

$Input_B$ = **1**

Output = **0**

Collecting and tabulating these results into a truth table, we see that the pattern matches that of the NAND gate:

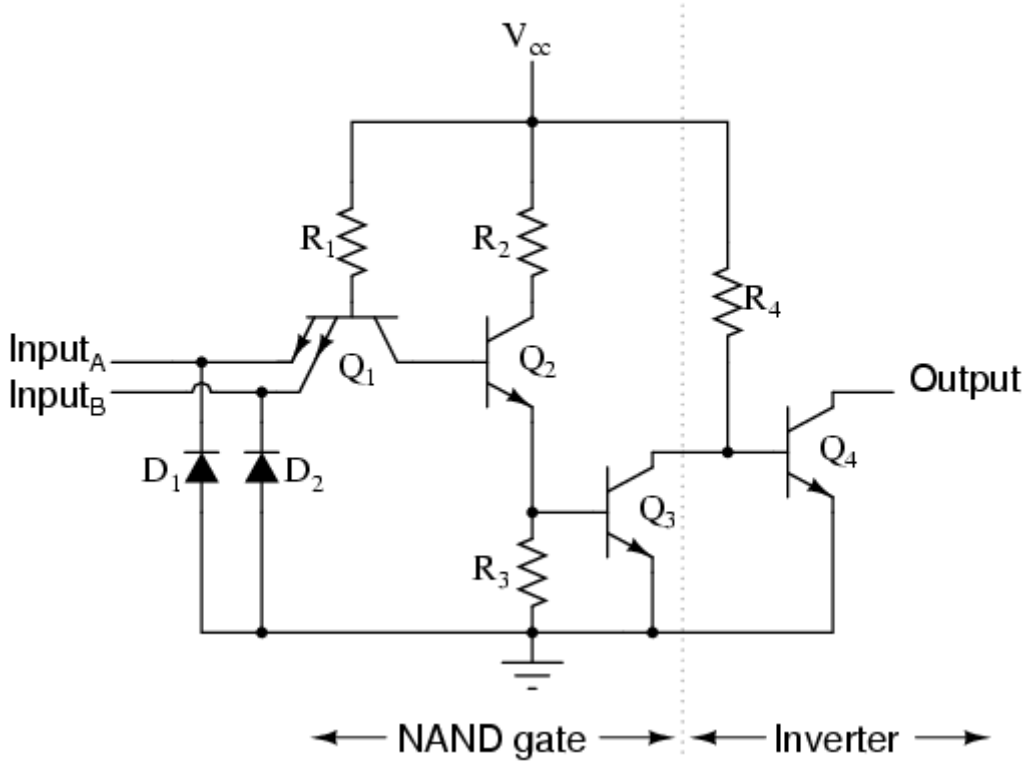
NAND gate



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

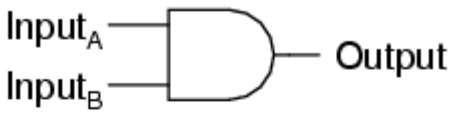
In the earlier section on NAND gates, this type of gate was created by taking an AND gate and increasing its complexity by adding an inverter (NOT gate) to the output. However, when we examine this circuit, we see that the NAND function is actually the simplest, most natural mode of operation for this TTL design. To create an AND function using TTL circuitry, we need to *increase* the complexity of this circuit by adding an inverter stage to the output, just like we had to add an additional transistor stage to the TTL inverter circuit to turn it into a buffer:

AND gate with open-collector output



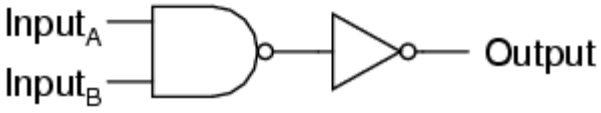
The truth table and equivalent gate circuit (an inverted-output NAND gate) are shown here:

AND gate



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Equivalent circuit

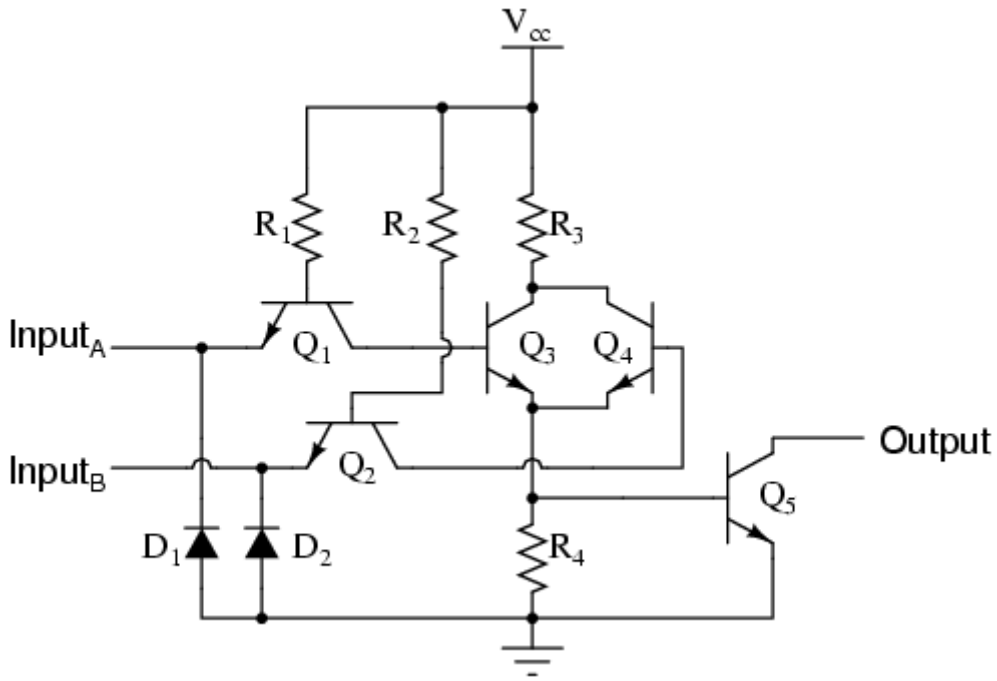


Of course, both NAND and AND gate circuits may be designed with totem-pole output stages rather than open-collector. I am opting to show the open-collector versions for the sake of simplicity.

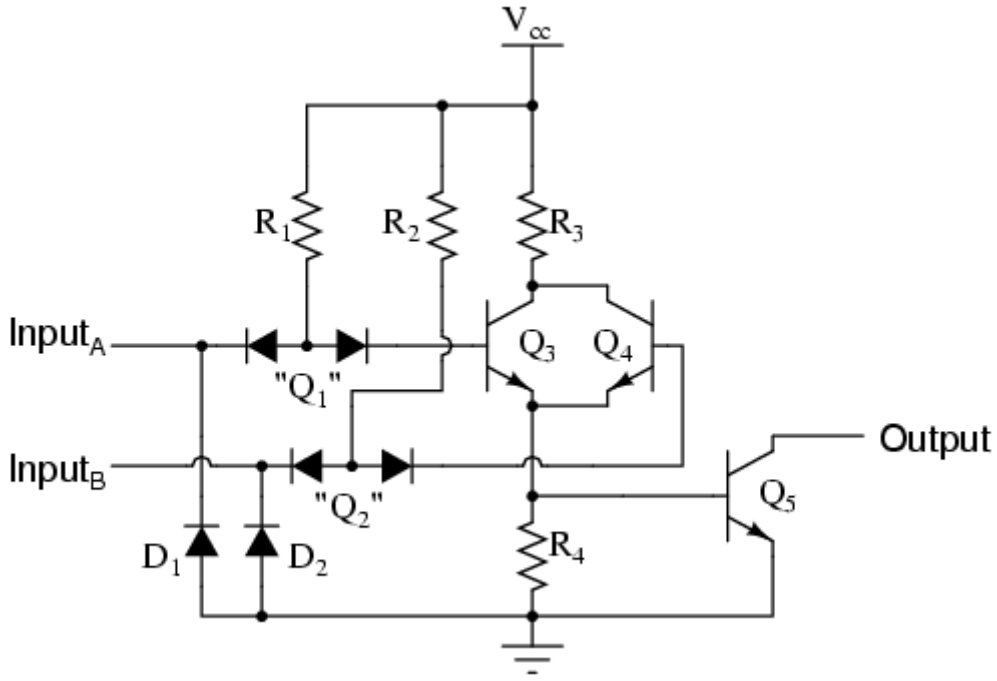
- **REVIEW:**
- A TTL NAND gate can be made by taking a TTL inverter circuit and adding another input.
- An AND gate may be created by adding an inverter stage to the output of the NAND gate circuit.

TTL NOR and OR gates

Let's examine the following TTL circuit and analyze its operation:



Transistors Q_1 and Q_2 are both arranged in the same manner that we've seen for transistor Q_1 in all the other TTL circuits. Rather than functioning as amplifiers, Q_1 and Q_2 are both being used as two-diode "steering" networks. We may replace Q_1 and Q_2 with diode sets to help illustrate:

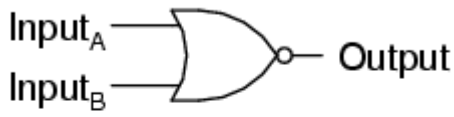


If input A is left floating (or connected to V_{cc}), current will go through the base of transistor Q_3, saturating it. If input A is grounded, that current is diverted away from Q_3's base through the left steering diode of "Q_1," thus forcing Q_3 into cutoff. The same can be said for input B and transistor Q_4: the logic level of input B determines Q_4's conduction: either saturated or cutoff.

Notice how transistors Q_3 and Q_4 are paralleled at their collector and emitter terminals. In essence, these two transistors are acting as paralleled switches, allowing current through resistors R_3 and R_4 according to the logic levels of inputs A and B. If *any* input is at a "high" (1) level, then at least one of the two transistors (Q_3 and/or Q_4) will be saturated, allowing current through resistors R_3 and R_4, and turning on the final output transistor Q_5 for a "low" (0) logic level output. The only way the output of this circuit can ever assume a "high" (1) state is if *both* Q_3 and Q_4 are cutoff, which means *both* inputs would have to be grounded, or "low" (0).

This circuit's truth table, then, is equivalent to that of the NOR gate:

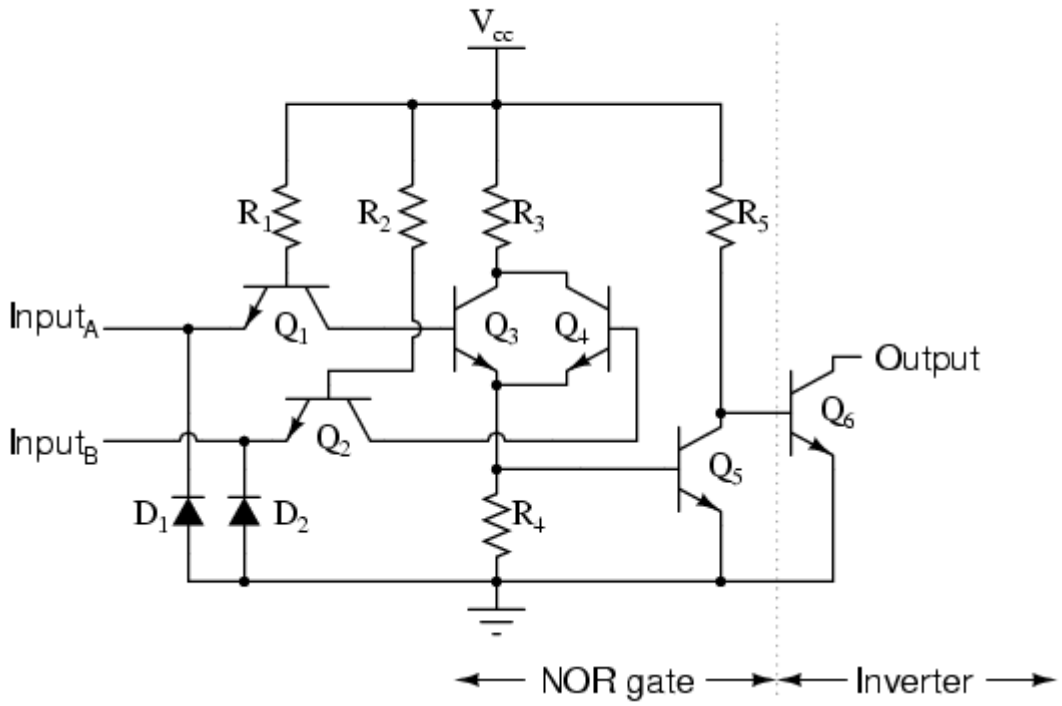
NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

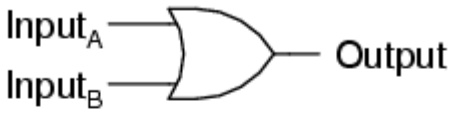
In order to turn this NOR gate circuit into an OR gate, we would have to invert the output logic level with another transistor stage, just like we did with the NAND-to-AND gate example:

OR gate with open-collector output



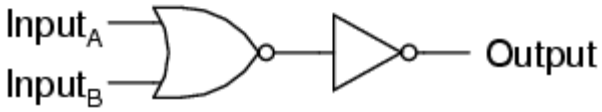
The truth table and equivalent gate circuit (an inverted-output NOR gate) are shown here:

OR gate



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Equivalent circuit



Of course, totem-pole output stages are also possible in both NOR and OR TTL logic circuits.

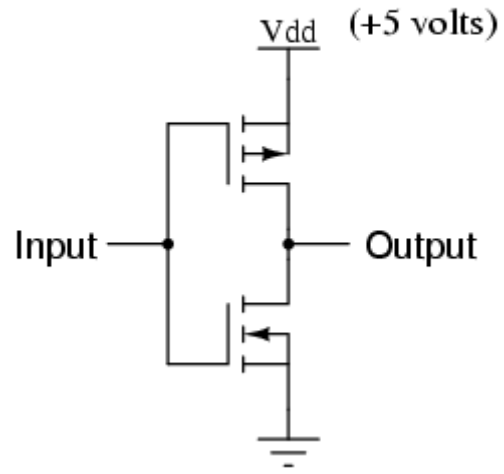
- **REVIEW:**
- An OR gate may be created by adding an inverter stage to the output of the NOR gate circuit.

CMOS gate circuitry

Up until this point, our analysis of transistor logic circuits has been limited to the *TTL* design paradigm, whereby bipolar transistors are used, and the general strategy of floating inputs being equivalent to "high" (connected to V_{cc}) inputs -- and correspondingly, the allowance of "open-collector" output stages -- is maintained. This, however, is not the only way we can build logic gates.

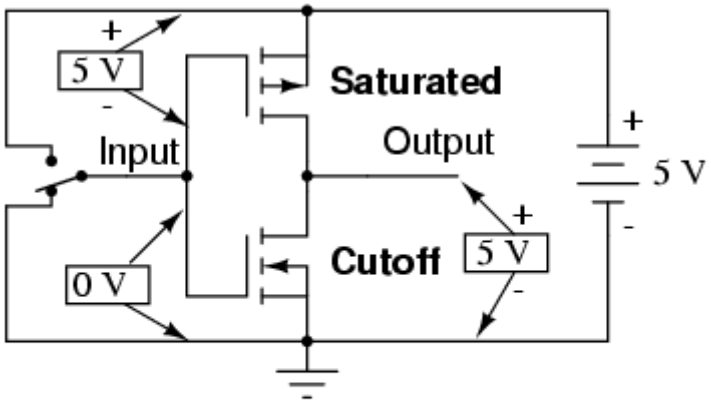
Field-effect transistors, particularly the insulated-gate variety, may be used in the design of gate circuits. Being voltage-controlled rather than current-controlled devices, IGFETs tend to allow very simple circuit designs. Take for instance, the following inverter circuit built using P- and N-channel IGFETs:

Inverter circuit using IGFETs



Notice the "V_{dd}" label on the positive power supply terminal. This label follows the same convention as "V_{cc}" in TTL circuits: it stands for the constant voltage applied to the drain of a field effect transistor, in reference to ground.

Let's connect this gate circuit to a power source and input switch, and examine its operation. Please note that these IGFET transistors are E-type (Enhancement-mode), and so are *normally-off* devices. It takes an applied voltage between gate and drain (actually, between gate and substrate) of the correct polarity to bias them *on*.



Input = "low" (0)

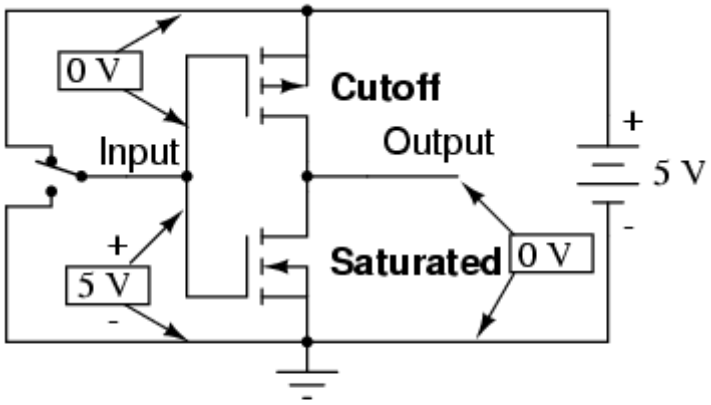
Output = "high" (1)

The upper transistor is a P-channel IGFET. When the channel (substrate) is made more positive than the gate (gate negative in reference to the substrate), the channel is enhanced and current is allowed between source and drain. So, in the above illustration, the top transistor is turned on.

The lower transistor, having zero voltage between gate and substrate (source), is in its normal mode: *off*. Thus, the action of these two transistors are such that the output terminal of the

gate circuit has a solid connection to V_{dd} and a very high resistance connection to ground. This makes the output "high" (1) for the "low" (0) state of the input.

Next, we'll move the input switch to its other position and see what happens:



Input = "high" (1)

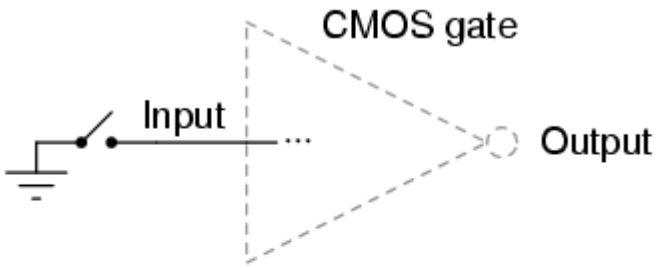
Output = "low" (0)

Now the lower transistor (N-channel) is saturated because it has sufficient voltage of the correct polarity applied between gate and substrate (channel) to turn it on (positive on gate, negative on the channel). The upper transistor, having zero voltage applied between its gate and substrate, is in its normal mode: *off*. Thus, the output of this gate circuit is now "low" (0). Clearly, this circuit exhibits the behavior of an inverter, or NOT gate.

Using field-effect transistors instead of bipolar transistors has greatly simplified the design of the inverter gate. Note that the output of this gate never floats as is the case with the simplest TTL circuit: it has a natural "totem-pole" configuration, capable of both sourcing and sinking load current. Key to this gate circuit's elegant design is the *complementary* use of both P- and N-channel IGFETs. Since IGFETs are more commonly known as MOSFETs (**M**etal-**O**xide-**S**emiconductor **F**ield **E**ffect **T**ransistor), and this circuit uses both P- and N-channel transistors together, the general classification given to gate circuits like this one is *CMOS*: **C**omplementary **M**etal **O**xide **S**emiconductor.

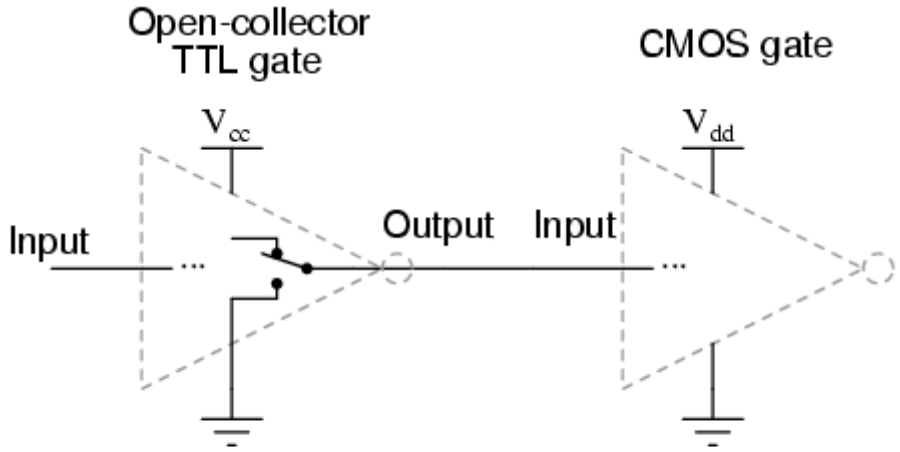
CMOS circuits aren't plagued by the inherent nonlinearities of the field-effect transistors, because as digital circuits their transistors always operate in either the *saturated* or *cutoff* modes and never in the *active* mode. Their inputs are, however, sensitive to high voltages generated by electrostatic (static electricity) sources, and may even be activated into "high" (1) or "low" (0) states by spurious voltage sources if left floating. For this reason, it is inadvisable to allow a CMOS logic gate input to float under any circumstances. Please note that this is very different from the behavior of a TTL gate where a floating input was safely interpreted as a "high" (1) logic level.

This may cause a problem if the input to a CMOS logic gate is driven by a single-throw switch, where one state has the input solidly connected to either V_{dd} or ground and the other state has the input floating (not connected to anything):



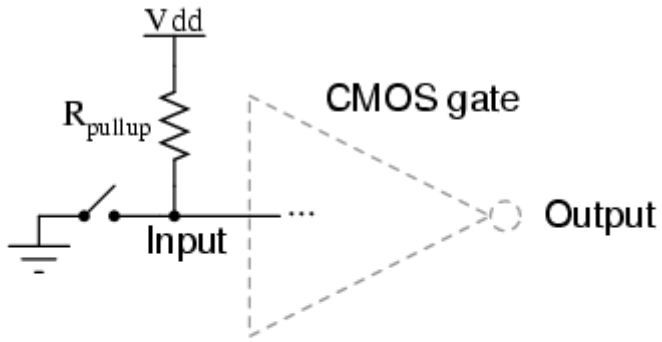
When switch is closed, the gate sees a definite "low" (0) input. However, when switch is open, the input logic level will be uncertain because it's floating.

Also, this problem arises if a CMOS gate input is being driven by an *open-collector* TTL gate. Because such a TTL gate's output floats when it goes "high" (1), the CMOS gate input will be left in an uncertain state:



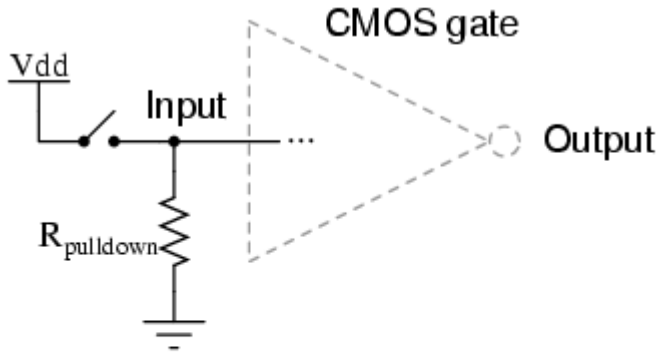
When the open-collector TTL gate's output is "high" (1), the CMOS gate's input will be left floating and in an uncertain logic state.

Fortunately, there is an easy solution to this dilemma, one that is used frequently in CMOS logic circuitry. Whenever a single-throw switch (or any other sort of gate output incapable of *both* sourcing and sinking current) is being used to drive a CMOS input, a resistor connected to either V_{dd} or ground may be used to provide a stable logic level for the state in which the driving device's output is floating. This resistor's value is not critical: 10 kΩ is usually sufficient. When used to provide a "high" (1) logic level in the event of a floating signal source, this resistor is known as a *pullup resistor*:



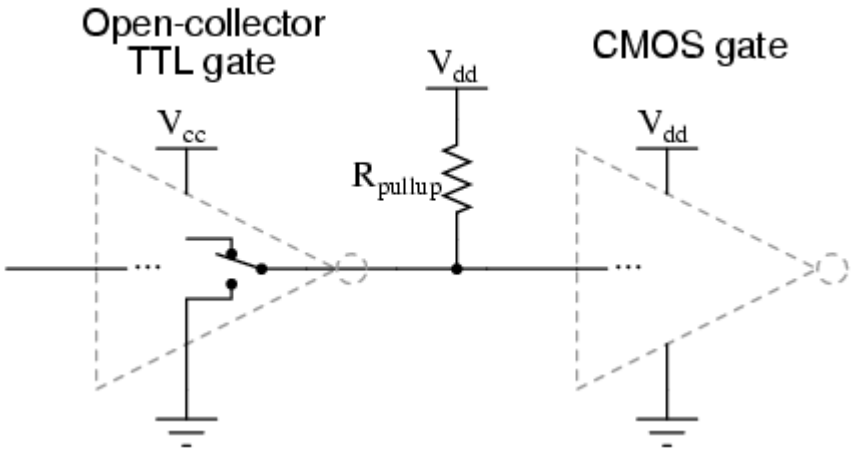
When switch is closed, the gate sees a definite "low" (0) input. When the switch is open, R_{pullup} will provide the connection to Vdd needed to secure a reliable "high" logic level for the CMOS gate input.

When such a resistor is used to provide a "low" (0) logic level in the event of a floating signal source, it is known as a *pulldown resistor*. Again, the value for a pulldown resistor is not critical:



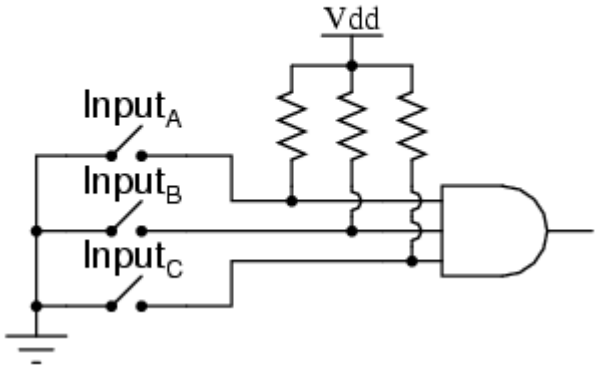
When switch is closed, the gate sees a definite "high" (1) input. When the switch is open, $R_{pulldown}$ will provide the connection to ground needed to secure a reliable "low" logic level for the CMOS gate input.

Because open-collector TTL outputs always sink, never source, current, pull*up* resistors are necessary when interfacing such an output to a CMOS gate input:



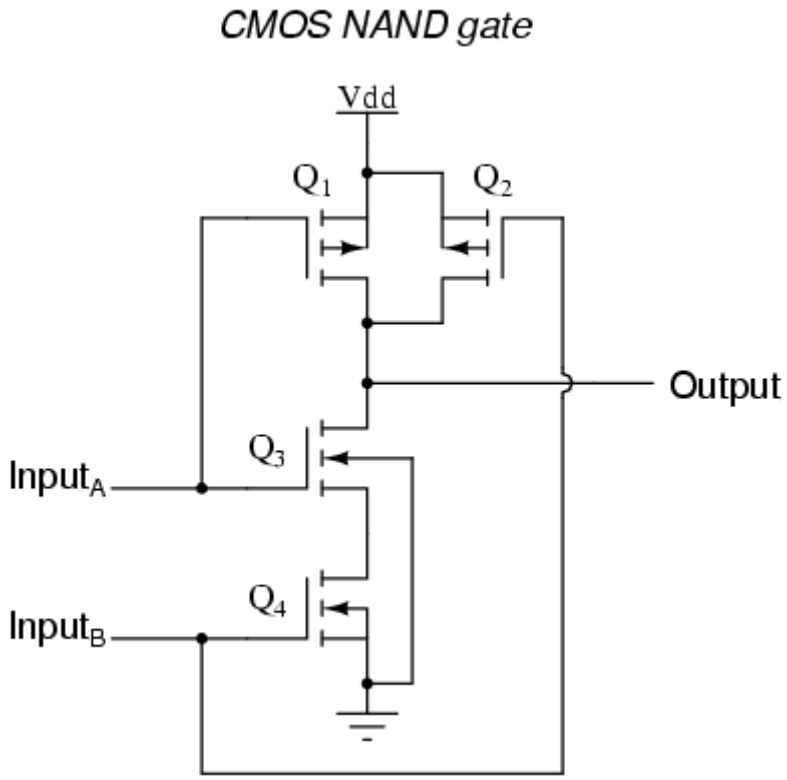
Although the CMOS gates used in the preceding examples were all inverters (single-input), the same principle of pullup and pulldown resistors applies to multiple-input CMOS gates. Of course, a separate pullup or pulldown resistor will be required for each gate input:

Pullup resistors for a 3-input CMOS AND gate

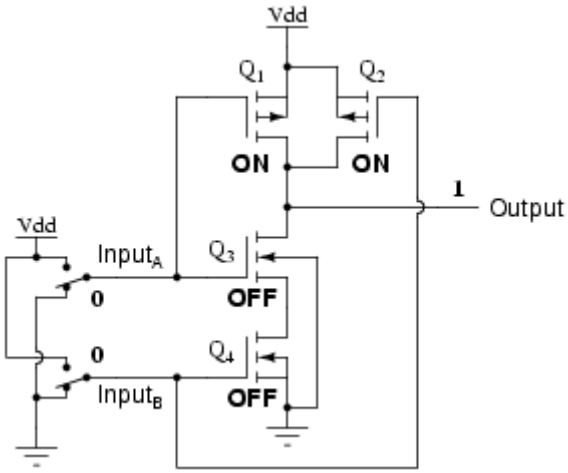


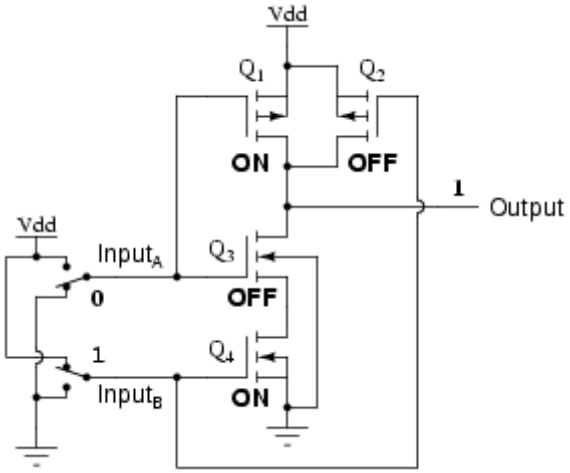
This brings us to the next question: how do we design multiple-input CMOS gates such as AND, NAND, OR, and NOR? Not surprisingly, the answer(s) to this question reveal a simplicity of design much like that of the CMOS inverter over its TTL equivalent.

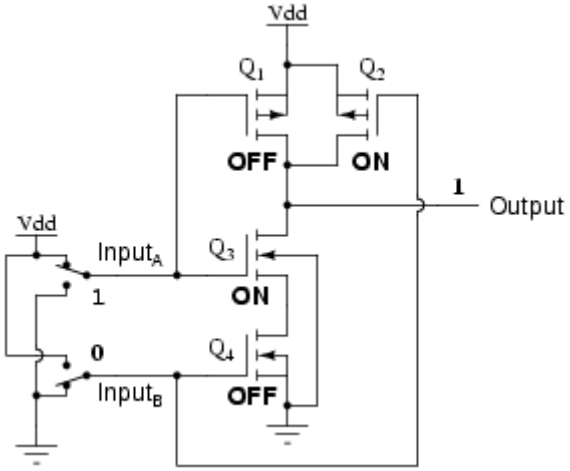
For example, here is the schematic diagram for a CMOS NAND gate:

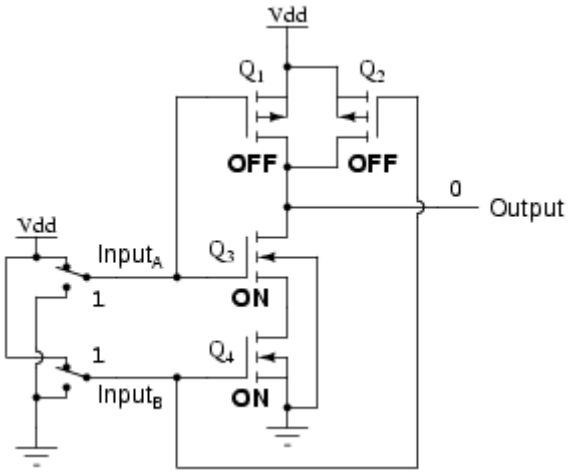


Notice how transistors Q_1 and Q_3 resemble the series-connected complementary pair from the inverter circuit. Both are controlled by the same input signal (input A), the upper transistor turning off and the lower transistor turning on when the input is "high" (1), and vice versa. Notice also how transistors Q_2 and Q_4 are similarly controlled by the same input signal (input B), and how they will also exhibit the same on/off behavior for the same input logic levels. The upper transistors of both pairs (Q_1 and Q_2) have their source and drain terminals paralleled, while the lower transistors (Q_3 and Q_4) are series-connected. What this means is that the output will go "high" (1) if *either* top transistor saturates, and will go "low" (0) only if *both* lower transistors saturate. The following sequence of illustrations shows the behavior of this NAND gate for all four possibilities of input logic levels (00, 01, 10, and 11):

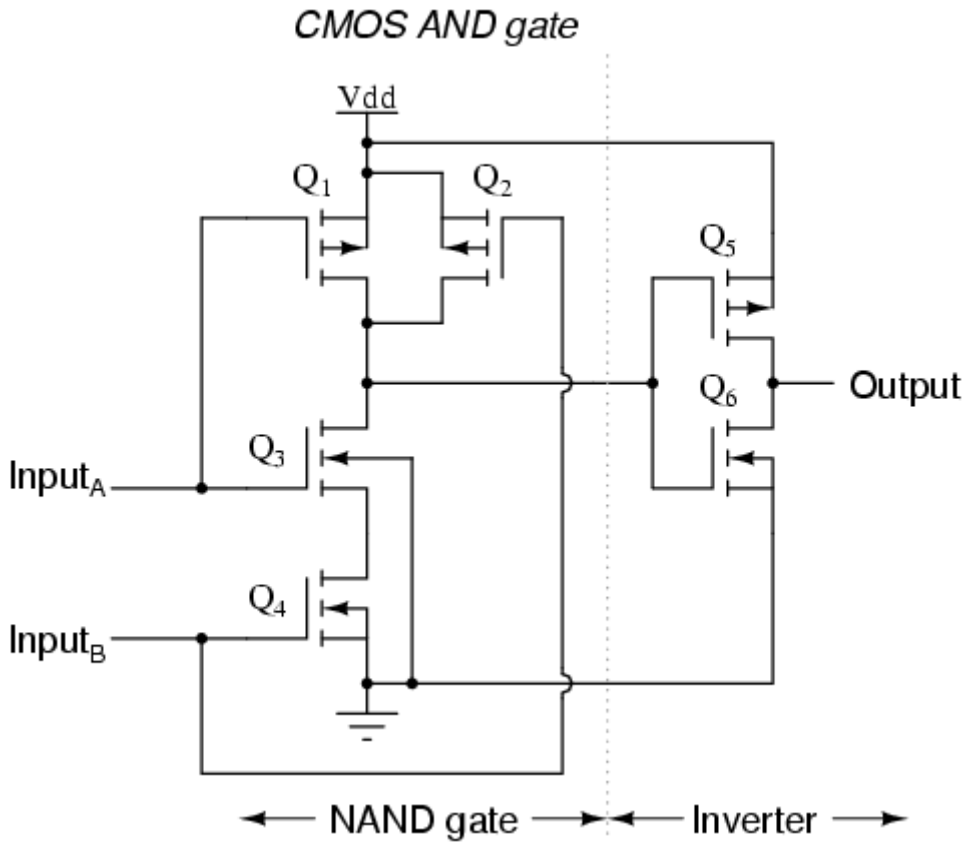




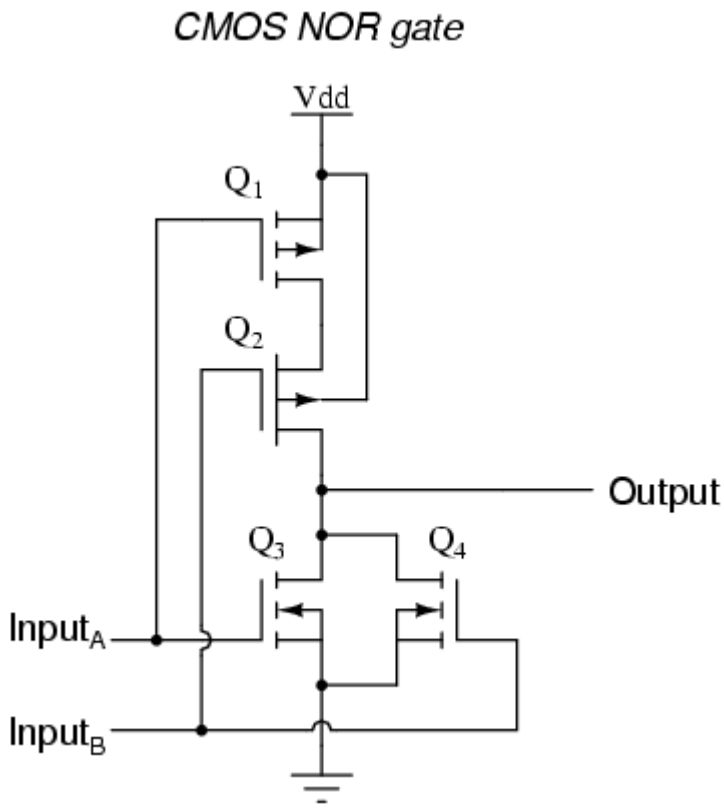




As with the TTL NAND gate, the CMOS NAND gate circuit may be used as the starting point for the creation of an AND gate. All that needs to be added is another stage of transistors to invert the output signal:

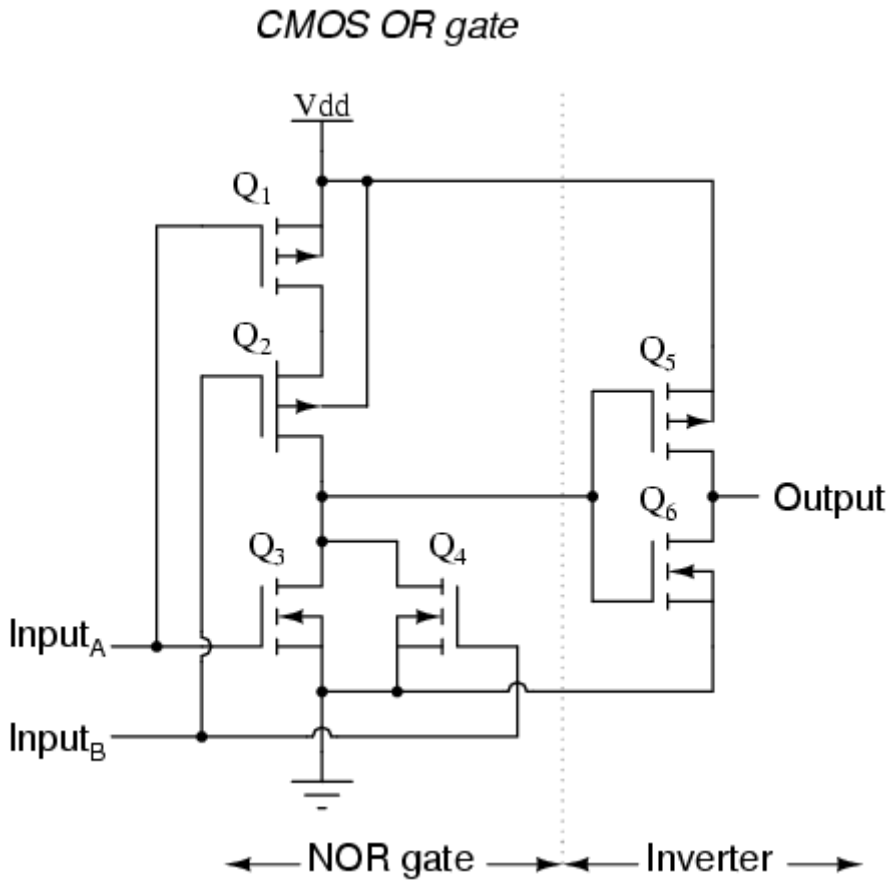


A CMOS NOR gate circuit uses four MOSFETs just like the NAND gate, except that its transistors are differently arranged. Instead of two paralleled *sourcing* (upper) transistors connected to V_{dd} and two series-connected *sinking* (lower) transistors connected to ground, the NOR gate uses two series-connected sourcing transistors and two parallel-connected sinking transistors like this:



As with the NAND gate, transistors Q_1 and Q_3 work as a complementary pair, as do transistors Q_2 and Q_4. Each pair is controlled by a single input signal. If *either* input A *or* input B are "high" (1), at least one of the lower transistors (Q_3 or Q_4) will be saturated, thus making the output "low" (0). Only in the event of *both* inputs being "low" (0) will both lower transistors be in cutoff mode and both upper transistors be saturated, the conditions necessary for the output to go "high" (1). This behavior, of course, defines the NOR logic function.

The OR function may be built up from the basic NOR gate with the addition of an inverter stage on the output:



Since it appears that any gate possible to construct using TTL technology can be duplicated in CMOS, why do these two "families" of logic design still coexist? The answer is that both TTL and CMOS have their own unique advantages.

First and foremost on the list of comparisons between TTL and CMOS is the issue of power consumption. In this measure of performance, CMOS is the unchallenged victor. Because the complementary P- and N-channel MOSFET pairs of a CMOS gate circuit are (ideally) never conducting at the same time, there is little or no current drawn by the circuit from the V_{dd} power supply except for what current is necessary to source current to a load. TTL, on the other hand, cannot function without some current drawn at all times, due to the biasing requirements of the bipolar transistors from which it is made.

There is a caveat to this advantage, though. While the power dissipation of a TTL gate remains rather constant regardless of its operating state(s), a CMOS gate dissipates more power as the frequency of its input signal(s) rises. If a CMOS gate is operated in a static (unchanging) condition, it dissipates zero power (ideally). However, CMOS gate circuits draw transient current during every output state switch from "low" to "high" and vice versa. So, the more often a CMOS gate switches modes, the more often it will draw current from the V_{dd} supply, hence greater power dissipation at greater frequencies.

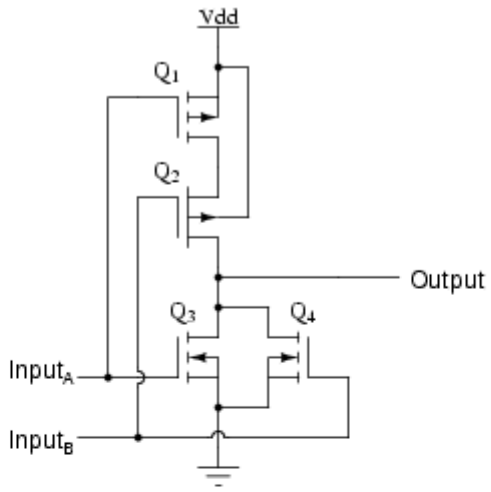
A CMOS gate also draws much less current from a driving gate output than a TTL gate because MOSFETs are voltage-controlled, not current-controlled, devices. This means that one gate can drive many more CMOS inputs than TTL inputs. The measure of how many gate inputs a single gate output can drive is called *fanout*.

Another advantage that CMOS gate designs enjoy over TTL is a much wider allowable range of power supply voltages. Whereas TTL gates are restricted to power supply (V_{cc}) voltages between 4.75 and 5.25 volts, CMOS gates are typically able to operate on any voltage between 3 and 15 volts! The reason behind this disparity in power supply voltages is the respective bias requirements of MOSFET versus bipolar junction transistors. MOSFETs are controlled exclusively by gate voltage (with respect to substrate), whereas BJTs are *current-controlled* devices. TTL gate circuit resistances are precisely calculated for proper bias currents assuming a 5 volt regulated power supply. Any significant variations in that power supply voltage will result in the transistor bias currents being incorrect, which then results in unreliable (unpredictable) operation. The only effect that variations in power supply voltage have on a CMOS gate is the voltage definition of a "high" (1) state. For a CMOS gate operating at 15 volts of power supply voltage (V_{dd}), an input signal must be close to 15 volts in order to be considered "high" (1). The voltage threshold for a "low" (0) signal remains the same: near 0 volts.

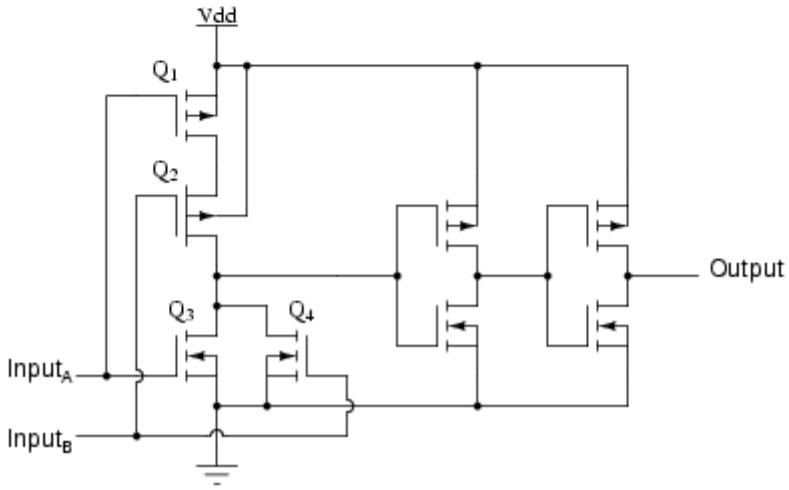
One decided disadvantage of CMOS is slow speed, as compared to TTL. The input capacitances of a CMOS gate are much, much greater than that of a comparable TTL gate -- owing to the use of MOSFETs rather than BJTs -- and so a CMOS gate will be slower to respond to a signal transition (low-to-high or vice versa) than a TTL gate, all other factors being equal. The RC time constant formed by circuit resistances and the input capacitance of the gate tend to impede the fast rise- and fall-times of a digital logic level, thereby degrading high-frequency performance.

A strategy for minimizing this inherent disadvantage of CMOS gate circuitry is to "buffer" the output signal with additional transistor stages, to increase the overall voltage gain of the device. This provides a faster-transitioning output voltage (high-to-low or low-to-high) for an input voltage slowly changing from one logic state to another. Consider this example, of an "unbuffered" NOR gate versus a "buffered," or *B-series*, NOR gate:

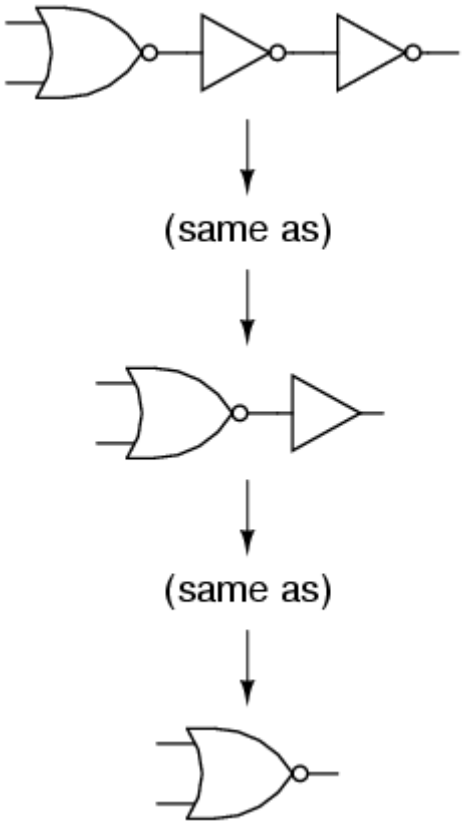
"Unbuffered" NOR gate



"B-series" (buffered) NOR gate



In essence, the B-series design enhancement adds two inverters to the output of a simple NOR circuit. This serves no purpose as far as digital logic is concerned, since two cascaded inverters simply cancel:



However, adding these inverter stages to the circuit does serve the purpose of increasing overall voltage gain, making the output more sensitive to changes in input state, working to overcome the inherent slowness caused by CMOS gate input capacitance.

- **REVIEW:**
- CMOS logic gates are made of IGFET (MOSFET) transistors rather than bipolar junction transistors.
- CMOS gate inputs are sensitive to static electricity. They may be damaged by high voltages, and they may assume any logic level if left floating.
- *Pullup* and *pulldown* resistors are used to prevent a CMOS gate input from floating if being driven by a signal source capable only of sourcing or sinking current.
- CMOS gates dissipate far less power than equivalent TTL gates, but their power dissipation increases with signal frequency, whereas the power dissipation of a TTL gate is approximately constant over a wide range of operating conditions.
- CMOS gate inputs draw far less current than TTL inputs, because MOSFETs are voltage-controlled, not current-controlled, devices.
- CMOS gates are able to operate on a much wider range of power supply voltages than TTL: typically 3 to 15 volts versus 4.75 to 5.25 volts for TTL.
- CMOS gates tend to have a much lower maximum operating frequency than TTL gates due to input capacitances caused by the MOSFET gates.
- *B-series* CMOS gates have "buffered" outputs to increase voltage gain from input to output, resulting in faster output response to input signal changes. This helps overcome the inherent slowness of CMOS gates due to MOSFET input capacitance and the RC time constant thereby engendered.

9.SHIFT REGISTERS

Introduction

Shift registers, like counters, are a form of *sequential logic*. Sequential logic, unlike combinational logic is not only affected by the present inputs, but also, by the prior history. In other words, sequential logic remembers past events.

Shift registers produce a discrete delay of a digital signal or waveform. A waveform synchronized to a *clock*, a repeating square wave, is delayed by **"n"** discrete clock times, where **"n"** is the number of shift register stages. Thus, a four stage shift register delays "data in" by four clocks to "data out". The stages in a shift register are *delay stages*, typically type **"D"** Flip-Flops or type **"JK"** Flip-flops.

Formerly, very long (several hundred stages) shift registers served as digital memory. This obsolete application is reminiscent of the acoustic mercury delay lines used as early computer memory.

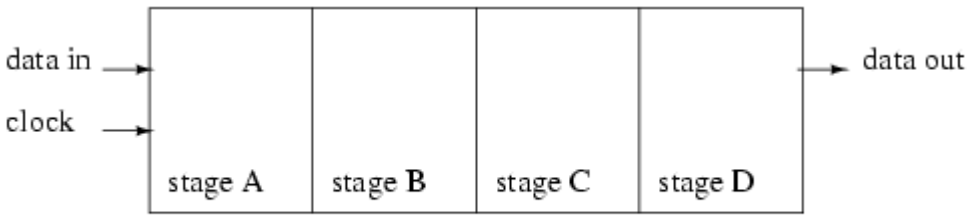
Serial data transmission, over a distance of meters to kilometers, uses shift registers to convert parallel data to serial form. Serial data communications replaces many slow parallel data wires with a single serial high speed circuit.

Serial data over shorter distances of tens of centimeters, uses shift registers to get data into and out of microprocessors. Numerous peripherals, including analog to digital converters, digital to analog converters, display drivers, and memory, use shift registers to reduce the amount of wiring in circuit boards.

Some specialized counter circuits actually use shift registers to generate repeating waveforms. Longer shift registers, with the help of feedback generate patterns so long that they look like random noise, *pseudo-noise*.

Basic shift registers are classified by structure according to the following types:

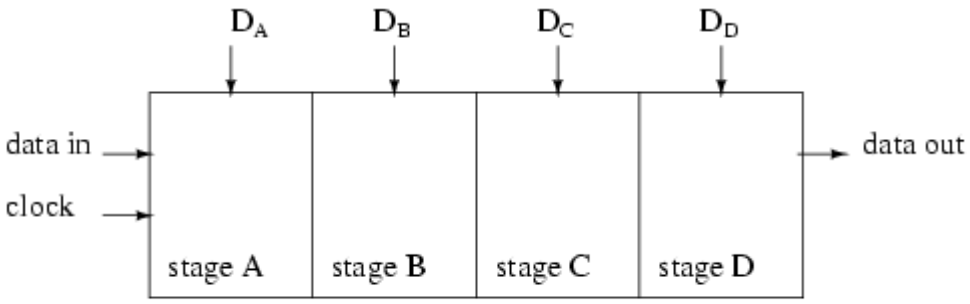
- Serial-in/serial-out
- Serial-in/parallel-out
- Parallel-in/serial-out
- Universal parallel-in/parallel-out
- Ring counter



Serial-in, serial-out shift register with 4-stages

Above we show a block diagram of a serial-in/serial-out shift register, which is 4-stages long. Data at the input will be delayed by four clock periods from the input to the output of the shift register.

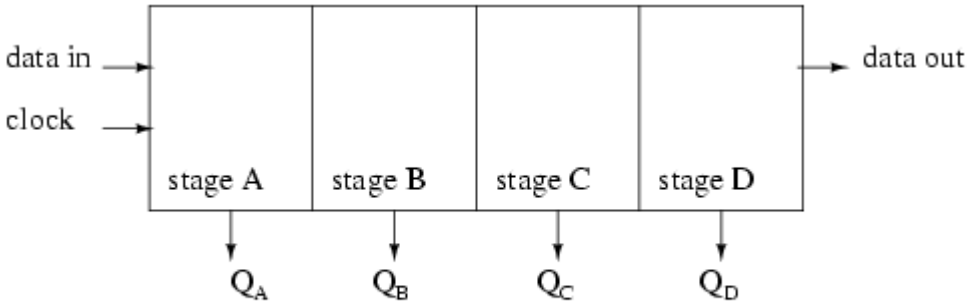
Data at "data in", above, will be present at the Stage **A** output after the first clock pulse. After the second pulse stage **A** data is transfered to stage **B** output, and "data in" is transfered to stage **A** output. After the third clock, stage **C** is replaced by stage **B**; stage **B** is replaced by stage **A**; and stage A is replaced by "data in". After the fourth clock, the data originally present at "data in" is at stage **D**, "output". The "first in" data is "first out" as it is shifted from "data in" to "data out".



Parallel-in, serial-out shift register with 4-stages

Data is loaded into all stages at once of a parallel-in/serial-out shift register. The data is then shifted out via "data out" by clock pulses. Since a 4- stage shift register is shown above, four clock pulses are required to shift out all of the data. In the diagram above, stage **D** data will be present at the "data out" up until the first clock pulse; stage **C** data will be present at "data out" between the first clock and the second clock pulse; stage **B** data will be present between the second clock and the third clock; and stage **A** data will be present between the third and the fourth clock. After the fourth clock pulse and thereafter, successive bits of "data in" should appear at "data out" of the shift register after a delay of four clock pulses.

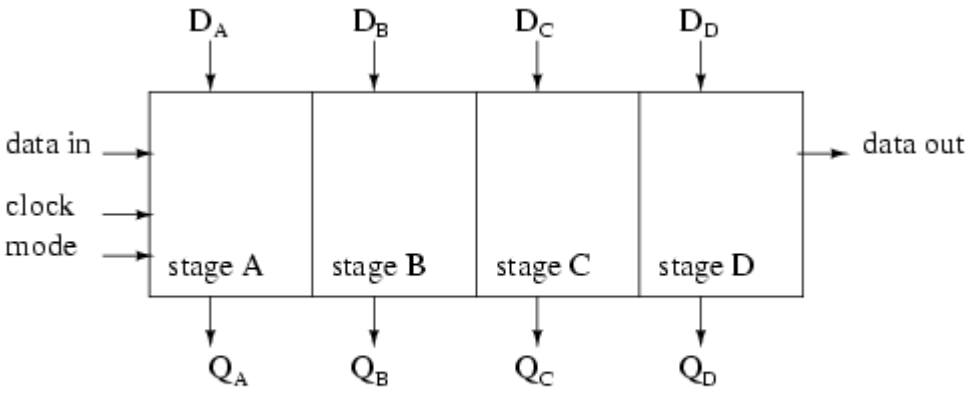
If four switches were connected to D_A through D_D, the status could be read into a microprocessor using only one data pin and a clock pin. Since adding more switches would require no additional pins, this approach looks attractive for many inputs.



Parallel-in, serial-out shift register with 4-stages

Above, four data bits will be shifted in from "data in" by four clock pulses and be available at Q_A through Q_D for driving external circuitry such as LEDs, lamps, relay drivers, and horns.

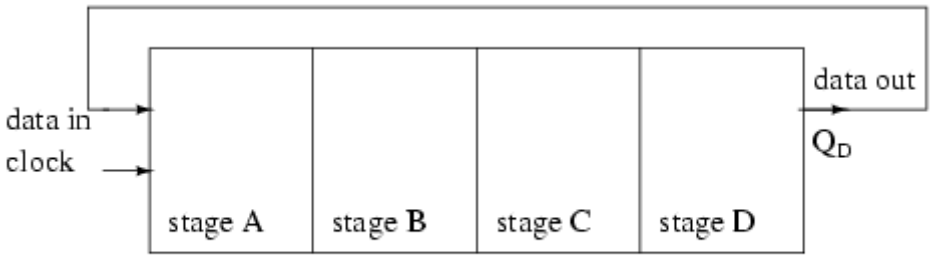
After the first clock, the data at "data in" appears at Q_A. After the second clock, The old Q_A data appears at Q_B; Q_A receives next data from "data in". After the third clock, Q_B data is at Q_C. After the fourth clock, Q_C data is at Q_D. This sat the data first present at "data in". The shift register should now contain four data bits.



Parallel-in, parallel-out shift register with 4-stages

A parallel-in/laralel-out shift register combines the function of the parallel-in, serial-out shift register with the function of the serial-in, parallel-out shift register to yields the universal shift register. The "do anything" shifter comes at a price-- the increased number of I/O (Input/Output) pins may reduce the number of stages which can be packaged.

Data presented at D_A through D_D is parallel loaded into the registers. This data at Q_A through Q_D may be shifted by the number of pulses presented at the clock input. The shifted data is available at Q_A through Q_D. The "mode" input, which may be more than one input, controls parallel loading of data from D_A through D_D, shifting of data, and the direction of shifting. There are shift registers which will shift data either left or right.



Ring Counter, shift register output fed back to input

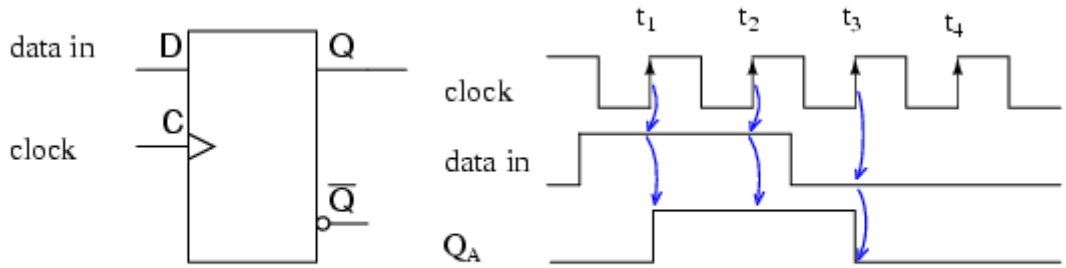
If the serial output of a shift register is connected to the serial input, data can be perpetually shifted around the ring as long as clock pulses are present. If the output is inverted before being fed back as shown above, we do not have to worry about loading the initial data into the "ring counter".

---

shift register, serial-in/serial-out shift

Serial-in, serial-out shift registers delay data by one clock time for each stage. They will store a bit of data for each register. A serial-in, serial-out shift register may be one to 64 bits in length, longer if registers or packages are cascaded.

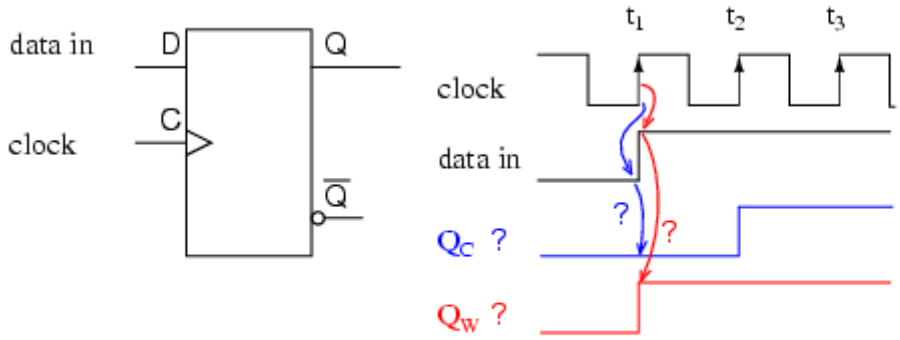
Below is a single stage shift register receiving data which is not synchronized to the register clock. The "data in" at the **D** pin of the type **D FF** (Flip-Flop) does not change levels when the clock changes for low to high. We may want to synchronize the data to a system wide clock in a circuit board to improve the reliability of a digital logic circuit.



Data present at clock time is transfered from D to Q.

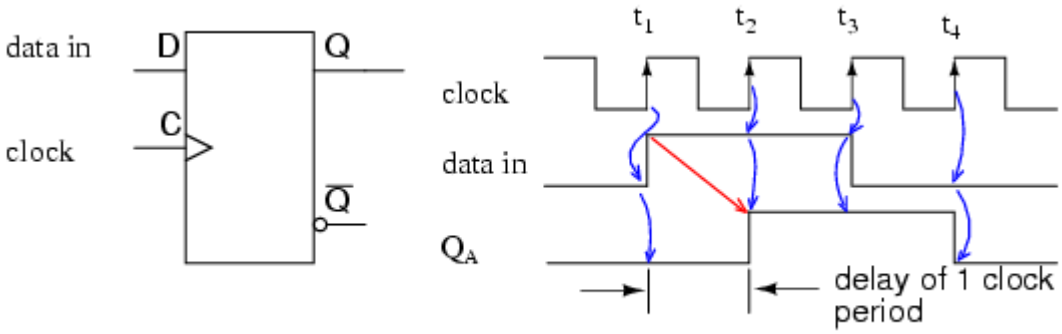
The obvious point (as compared to the figure below) illustrated above is that whatever "data in" is present at the **D** pin of a type **D** FF is transfered from D to output Q at clock time. Since our example shift register uses positive edge sensitive storage elements, the output **Q** follows

the **D** input when the clock transitions from low to high as shown by the up arrows on the diagram above. There is no doubt what logic level is present at clock time because the data is stable well before and after the clock edge. This is seldom the case in multi-stage shift registers. But, this was an easy example to start with. We are only concerned with the positive, low to high, clock edge. The falling edge can be ignored. It is very easy to see **Q** follow **D** at clock time above. Compare this to the diagram below where the "data in" appears to change with the positive clock edge.



Does the clock t_1 see a 0 or a 1 at data in at D? Which output is correct, Q_C or Q_W ?

Since "data in" appears to changes at clock time t_1 above, what does the type **D** FF see at clock time? The short over simplified answer is that it sees the data that was present at **D** prior to the clock. That is what is transfered to **Q** at clock time t_1. The correct waveform is Q_C. At t_1 Q goes to a zero if it is not already zero. The **D** register does not see a one until time t_2, at which time Q goes high.

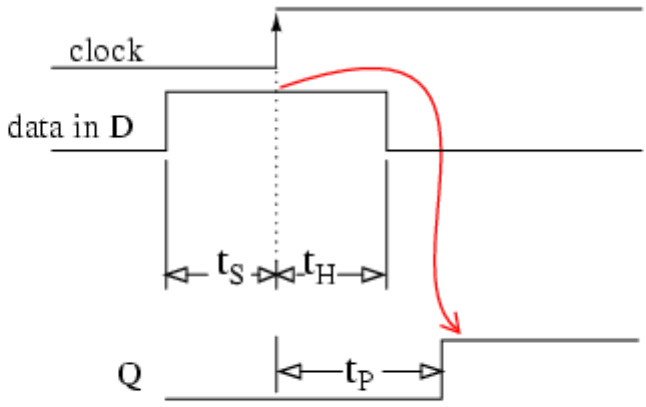


Data present t_H before clock time at D is transfered to Q.

Since data, above, present at **D** is clocked to **Q** at clock time, and **Q** cannot change until the next clock time, the **D** FF delays data by one clock period, provided that the data is already

synchronized to the clock. The Q_A waveform is the same as "data in" with a one clock period delay.

A more detailed look at what the input of the type **D** Flip-Flop sees at clock time follows. Refer to the figure below. Since "data in" appears to changes at clock time (above), we need further information to determine what the **D** FF sees. If the "data in" is from another shift register stage, another same type **D** FF, we can draw some conclusions based on *data sheet* information. Manufacturers of digital logic make available information about their parts in data sheets, formerly only available in a collection called a *data book*. Data books are still available; though, the manufacturer's web site is the modern source.



Data must be present (t_S) before the clock and after(t_H) the clock. Data is delayed from D to Q by propagation delay (t_P)

The following data was extracted from the CD4006b data sheet for operation at $5V_{DC}$, which serves as an example to illustrate timing.

[*]

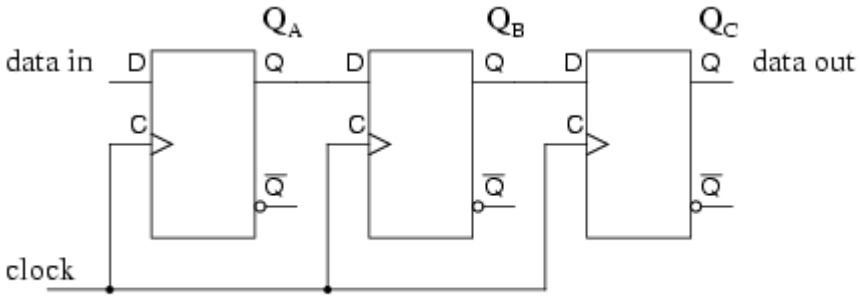
- t_S=100ns
- t_H=60ns
- t_P=200-400ns typ/max

t_S is the *setup time*, the time data must be present before clock time. In this case data must be present at **D** 100ns prior to the clock. Furthermore, the data must be held for *hold time* t_H=60ns after clock time. These two conditions must be met to reliably clock data from **D** to **Q** of the Flip-Flop.

There is no problem meeting the setup time of 60ns as the data at **D** has been there for the whole previous clock period if it comes from another shift register stage. For example, at a clock frequency of 1 Mhz, the clock period is 1000 µs, plenty of time. Data will actually be present for 1000µs prior to the clock, which is much greater than the minimum required t_S of 60ns.

The hold time t_H=60ns is met because D connected to Q of another stage cannot change any faster than the propagation delay of the previous stage t_P=200ns. Hold time is met as long as the propagation delay of the previous **D** FF is greater than the hold time. Data at **D** driven by another stage **Q** will not change any faster than 200ns for the CD4006b.

To summarize, output **Q** follows input D at nearly clock time if Flip-Flops are cascaded into a multi-stage shift register.



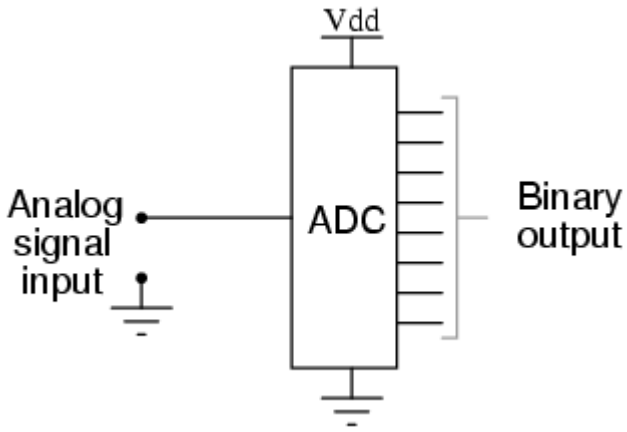
Serial-in, serial-out shift register using type "D" storage elements

10. DIGITAL-ANALOG CONVERSION

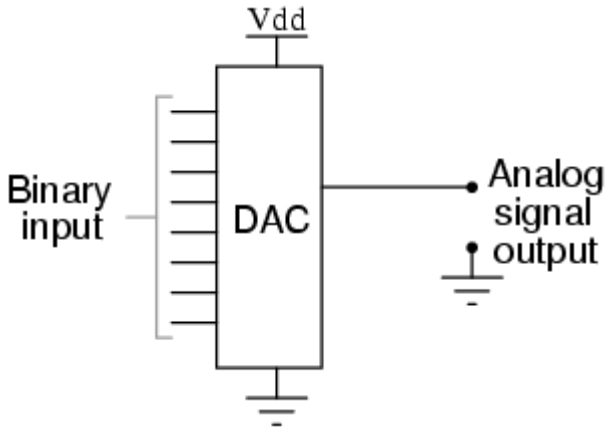
Introduction

Connecting digital circuitry to sensor devices is simple if the sensor devices are inherently digital themselves. Switches, relays, and encoders are easily interfaced with gate circuits due to the on/off nature of their signals. However, when analog devices are involved, interfacing becomes much more complex. What is needed is a way to electronically translate analog signals into digital (binary) quantities, and vice versa. An *analog-to-digital converter*, or ADC, performs the former task while a *digital-to-analog converter*, or DAC, performs the latter.

An ADC inputs an analog electrical signal such as voltage or current and outputs a binary number. In block diagram form, it can be represented as such:

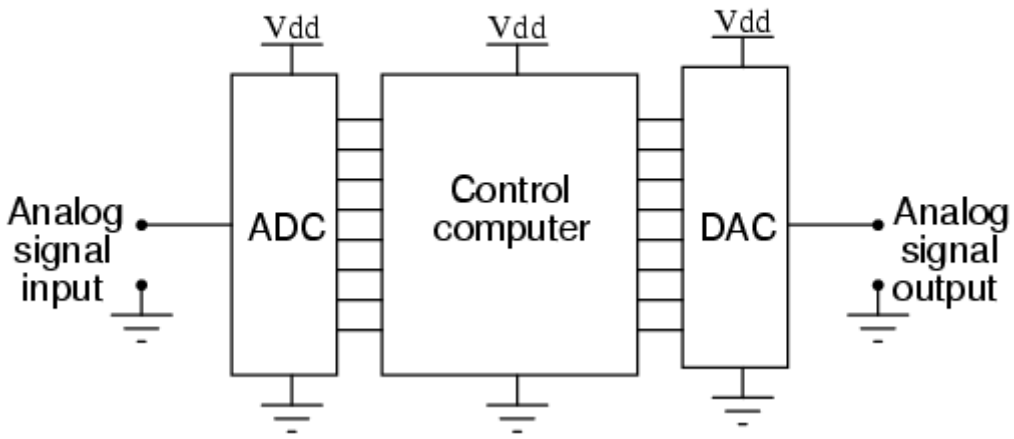


A DAC, on the other hand, inputs a binary number and outputs an analog voltage or current signal. In block diagram form, it looks like this:



Together, they are often used in digital systems to provide complete interface with analog sensors and output devices for control systems such as those used in automotive engine controls:

Digital control system with analog I/O

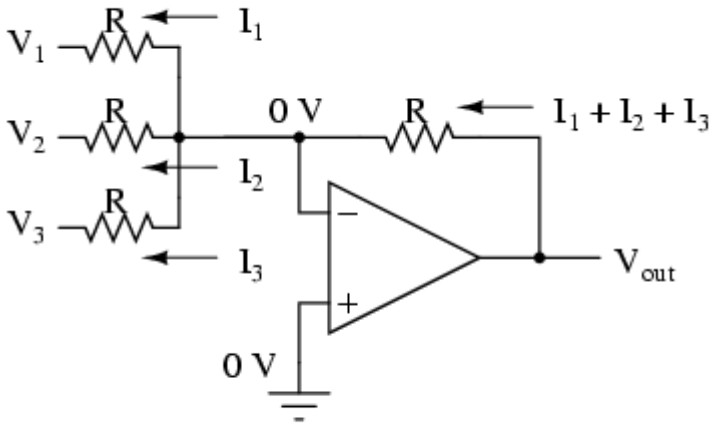


It is much easier to convert a digital signal into an analog signal than it is to do the reverse. Therefore, we will begin with DAC circuitry and then move to ADC circuitry.

The R/2^{n}R DAC

This DAC circuit, otherwise known as the *binary-weighted-input* DAC, is a variation on the inverting summer op-amp circuit. If you recall, the classic inverting summer circuit is an operational amplifier using negative feedback for controlled gain, with several voltage inputs and one voltage output. The output voltage is the inverted (opposite polarity) sum of all input voltages:

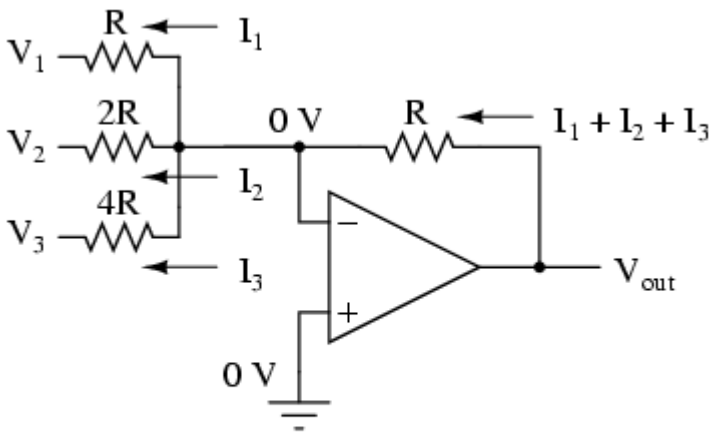
Inverting summer circuit



$$\mathbf{V_{out} = - (V_1 + V_2 + V_3)}$$

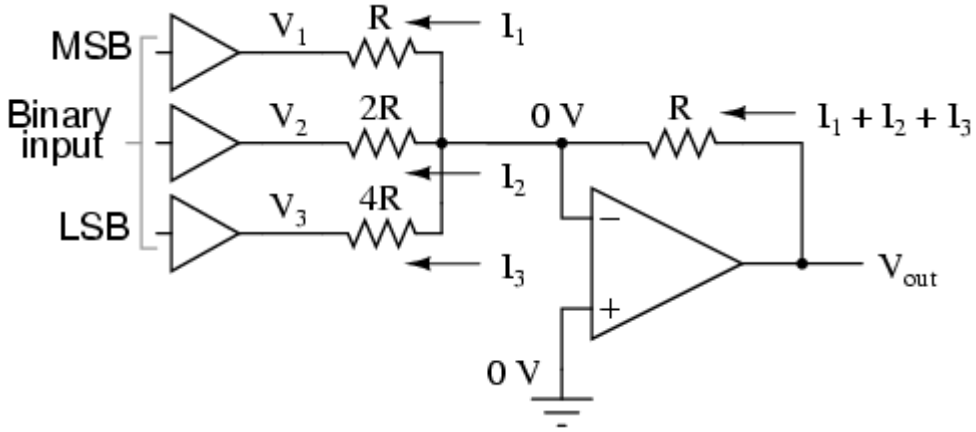
For a simple inverting summer circuit, all resistors must be of equal value. If any of the input resistors were different, the input voltages would have different degrees of effect on the output, and the output voltage would not be a true sum. Let's consider, however, intentionally

setting the input resistors at different values. Suppose we were to set the input resistor values at multiple powers of two: R, 2R, and 4R, instead of all the same value R:



$$V_{out} = -\left(V_1 + \frac{V_2}{2} + \frac{V_3}{4}\right)$$

Starting from V_1 and going through V_3, this would give each input voltage exactly half the effect on the output as the voltage before it. In other words, input voltage V_1 has a 1:1 effect on the output voltage (gain of 1), while input voltage V_2 has half that much effect on the output (a gain of 1/2), and V_3 half of that (a gain of 1/4). These ratios are were not arbitrarily chosen: they are the same ratios corresponding to place weights in the binary numeration system. If we drive the inputs of this circuit with digital gates so that each input is either 0 volts or full supply voltage, the output voltage will be an analog representation of the binary value of these three bits.

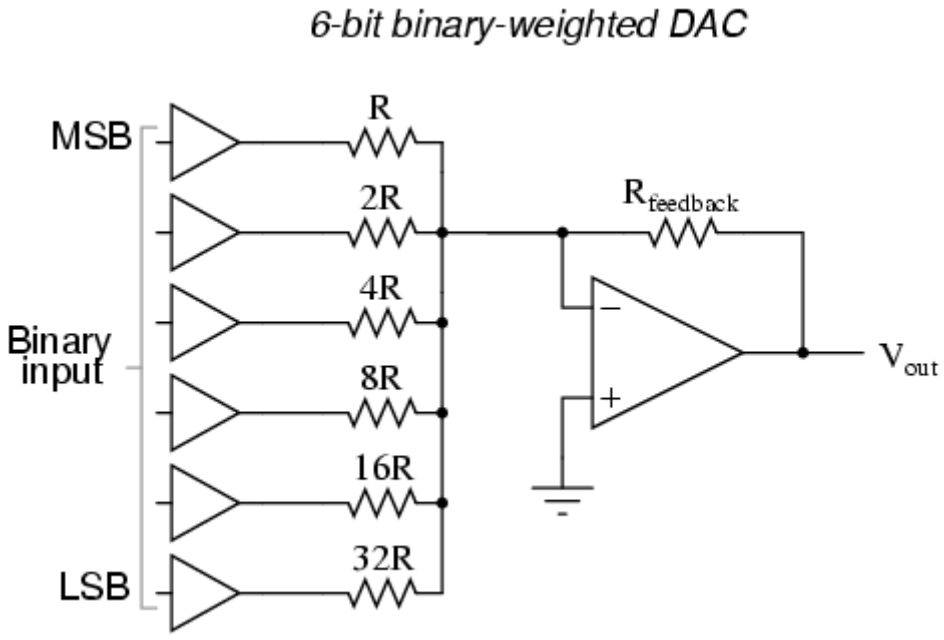


If we chart the output voltages for all eight combinations of binary bits (000 through 111) input to this circuit, we will get the following progression of voltages:

Binary	Output voltage
000	0.00 V
001	-1.25 V
010	-2.50 V
011	-3.75 V
100	-5.00 V
101	-6.25 V
110	-7.50 V
111	-8.75 V

Note that with each step in the binary count sequence, there results a 1.25 volt change in the output.

If we wish to expand the resolution of this DAC (add more bits to the input), all we need to do is add more input resistors, holding to the same power-of-two sequence of values:

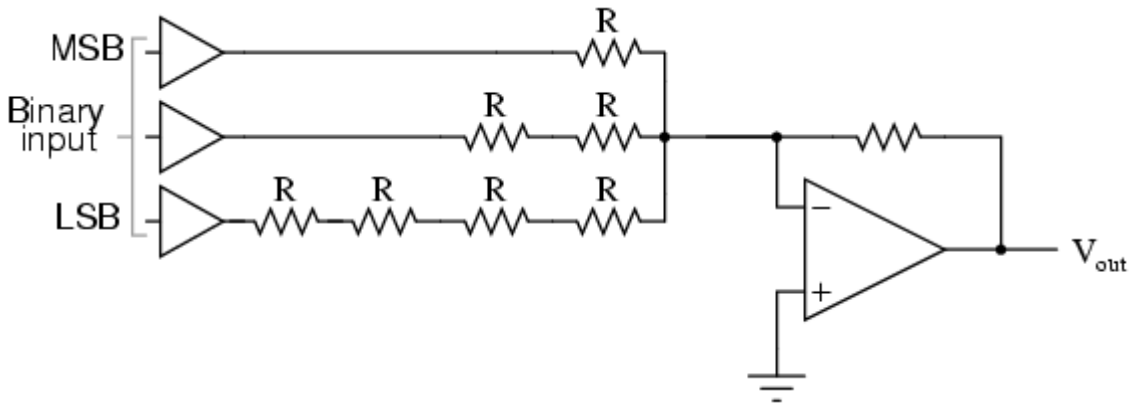


It should be noted that all logic gates must output exactly the same voltages when in the "high" state. If one gate is outputting +5.02 volts for a "high" while another is outputting only +4.86 volts, the analog output of the DAC will be adversely affected. Likewise, all "low" voltage levels should be identical between gates, ideally 0.00 volts exactly. It is recommended that CMOS output gates are used, and that input/feedback resistor values are chosen so as to minimize the amount of current each gate has to source or sink.

The R/2R DAC

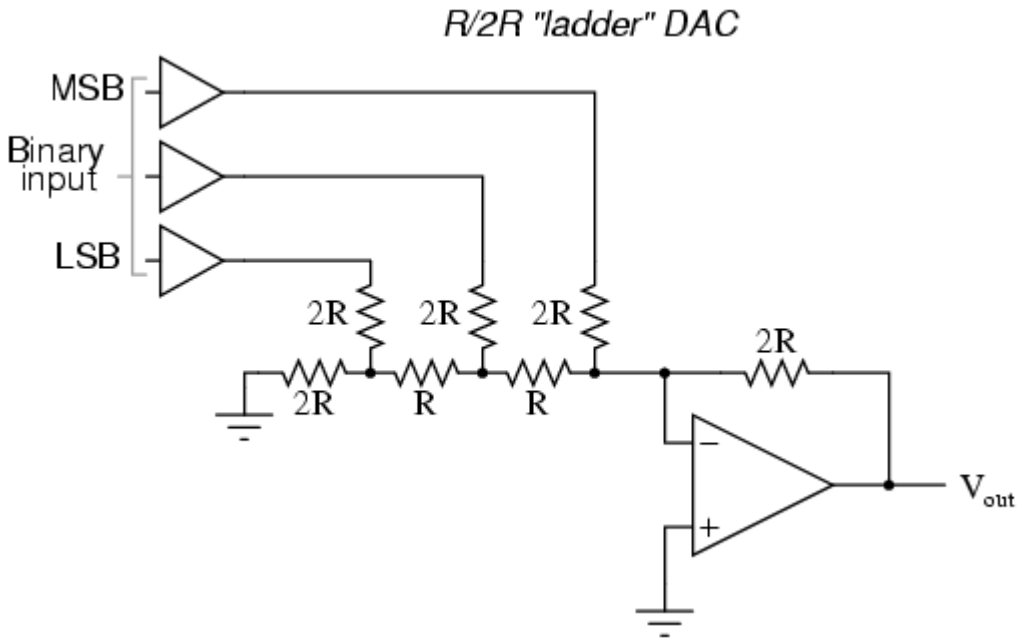
An alternative to the binary-weighted-input DAC is the so-called R/2R DAC, which uses fewer unique resistor values. A disadvantage of the former DAC design was its requirement of several different precise input resistor values: one unique value per binary input bit. Manufacture may be simplified if there are fewer different resistor values to purchase, stock, and sort prior to assembly.

Of course, we could take our last DAC circuit and modify it to use a single input resistance value, by connecting multiple resistors together in series:



Unfortunately, this approach merely substitutes one type of complexity for another: volume of components over diversity of component values. There is, however, a more efficient design methodology.

By constructing a different kind of resistor network on the input of our summing circuit, we can achieve the same kind of binary weighting with only two kinds of resistor values, and with only a modest increase in resistor count. This "ladder" network looks like this:



Mathematically analyzing this ladder network is a bit more complex than for the previous circuit, where each input resistor provided an easily-calculated gain for that bit. For those who are interested in pursuing the intricacies of this circuit further, you may opt to use Thevenin's theorem for each binary input (remember to consider the effects of the *virtual ground*), and/or use a simulation program like SPICE to determine circuit response. Either way, you should obtain the following table of figures:

Binary	Output voltage
000	0.00 V
001	-1.25 V
010	-2.50 V
011	-3.75 V
100	-5.00 V
101	-6.25 V
110	-7.50 V
111	-8.75 V

As was the case with the binary-weighted DAC design, we can modify the value of the feedback resistor to obtain any "span" desired. For example, if we're using +5 volts for a "high" voltage level and 0 volts for a "low" voltage level, we can obtain an analog output directly corresponding to the binary input (011 = -3 volts, 101 = -5 volts, 111 = -7 volts, etc.) by using a feedback resistance with a value of 1.6R instead of 2R.