Chapter 9: Digital Electronics I – Static Logic

Digital electronics is a system that operates in a finite number of states, usually two. I will only consider systems that are two state systems. The electronic circuits in the system will either be in a Hi state or a Lo state, usually designated as 1 or 0. The Hi and Lo states are called logic states because they are either True (Hi) or False (Lo). The actual voltages associated with the high or low state vary from system to system or logic family to logic family. (In the TTL family of devices, a 0 or low state is any voltage between 0V and 0.8V while the high state or 1 is any voltage between 2.4V and 5V.)

A. Basic Digital Logic

Basic digital circuits perform logical (mathematical) operations on digital signals. For this section, think of 0 as the low state and 1 as the high state. The 0 and 1 are NOT 0V and 1V. They are the symbols for the Hi and Lo states. The simplest operation is the INVERT. If a digital signal is 1 and it is connected to a circuit (usually called a **GATE**) that performs an invert operation, the output of the circuit or gate is a 0. When describing the action of a gate one usually uses a **TRUTH TABLE** which show the output for all possible input(s). For an INVERT operation there is only one input so the truth table is shown below as is the symbol for an invert operation. The input is labeled A and the output is usually y.

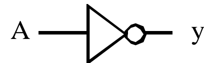

Fig 9.1 An Inverter

Input (A)	Output ($Y = \overline{A}$)
0	1
1	0

The triangle is the symbol for a “buffer” and the open circle, o, indicates the inversion. The open circle is often called a bubble. A triangle without the bubble would not invert the signal an in that case, y = A instead of the inverse of A, usually written $\overline{A}$, where the ¯ indicates inversion.

There are several basic logic functions that operate on two (or more) inputs to produce a single output that depends on the state of the inputs. These are AND, OR, NOR, NAND, EXOR and EXNOR. At first we will consider gates with two inputs. The symbol and truth table for the AND gate are shown below. The logic is that both (i.e. ALL) inputs have to be 1 for the output to be 1. Otherwise the output is 0. It is <u>analogous</u> to multiplication. The output looks like the product of the inputs.

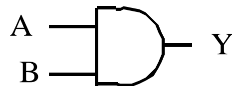

Fig. 9.2 An AND gate

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(Don’t push the analogy too far though.) The OR gate symbol and truth table are shown below. The logic is that if ANY input is 1, the output is 1.

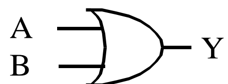

Fig. 9.3 An OR gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

The NAND and NOR gates (Not AND and Not OR) are shown below with their truth tables. The NAND gate is just an AND gate with its output inverted (indicated by the open circle or bubble). Similarly the NOR gate is an OR gate with its output inverted.

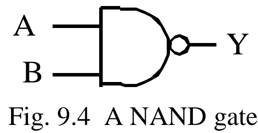

Fig. 9.4 A NAND gate

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

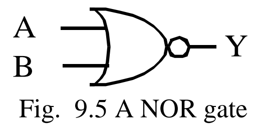

Fig. 9.5 A NOR gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

The Exclusive OR gate (EXOR) is shown below. The output is 1 if and only if the two inputs are different. If they are the same (0,0) or (1,1), the output is 0. The EXOR gate only has two inputs. The AND, OR, NAND, and NOR gates can have any number of inputs.

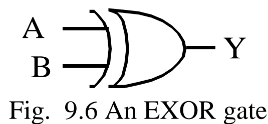

Fig. 9.6 An EXOR gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

The EXNOR is made by inverting the output of an EXOR gate.

You can make any of the above two input gates with by combining two input NAND gates. An inverter is just a NAND gate with the input going to both the A and B inputs. Consider the gate combinations below. You might try to find the truth table for this gate and compare it to the

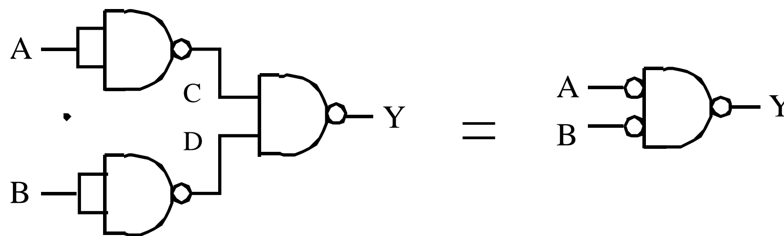

Fig. 9.7 Three NAND Gates

truth tables above. What simple two input gate is it equivalent to? When doing these I recommend that you make a truth table and then just fill it in.

In the above example I have labeled the inputs to the third NAND gate as C and D. Note that $C = \overline{A}$ and $D = \overline{B}$. I would make my truth table look like the one at the right. The

A	B	$C=\overline{A}$	$D=\overline{B}$	Y
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

Inclusion of the columns for C and D make it a little easier to find the output, Y. Looking at Y and comparing them to A and B shows that this is equivalent to an OR gate.

B. Binary Numbers

We use digital logic gates to perform numerical operations on binary numbers. In the binary system, each place holder, or bit, can have one of two values, 0 or 1. In the decimal system each place holder, or decimal, can have one of ten values, 0 through 9. In the decimal system a 4 digit number can be represented by (A_3,A_2,A_1,A_0) where each A can be 0-9. For the number 2435, $A_3 = 2$, $A_2 = 4$, $A_1 = 3$, and $A_0 = 5$. A_3 represents the number of thousands, 10^3, A_2 represents the number of hundreds, 10^2, A_1 represents the number of tens, 10^1, and A_0 represents the number of ones, 10^0. If I represent a four bit binary number by (A_3,A_2,A_1,A_0), then A_0 represents the number of ones, 2^0, A_1 represents the number of twos, 2^1, A_2 represents the number of fours, 2^2, and A_3 represents the number of eights, 2^3. Therefore the binary number (1010) is one 8 and one 2, or in decimal, 10_{10}, where the subscript after the 10 indicates that the number is a decimal or base ten representation since 10 could represent either ten in decimal or two in binary. If 10 is a binary I will often write it as 10_b to make it clear that it is a binary number. It is relatively easy to convert from binary to decimal, but slightly harder to convert from decimal to binary. In the four bit binary number by (A_3,A_2,A_1,A_0) we say that the number of ones, A_0, is the least significant bit in the number and A_3, the number of eights, is the most significant bit in the number.

To convert from decimal to binary, the preferred way is the successive division by two method. Take a decimal number and divide it by two. The remainder is the number of ones or the least significant bit in the equivalent binary number. Divide the quotient of the first division by two again, and the remainder for this division is the value of the next most significant bit. Keep repeating this until the final division by tow yields 0 with a remainder of 0 or 1. Consider the example below for the decimal number 27. (I'm using the term quotient to be the integer part of the division of two integers.)

	Quotient	Remainder
$27 \div 2 =$	13	$1 = A_0$
$12 \div 2 =$	6	$1 = A_1$
$6 \div 2 =$	3	$0 = A_2$
$3 \div 2 =$	2	$1 = A_3$
$2 \div 2 =$	1	$0 = A_4$
$1 \div 2 =$	0	$1 = A_5$

The result is the binary number 101011, a six bit binary number. I will usually write this a

10 1011

with a space between every four bits. It is like putting commas between large decimal numbers to make them more "readable". I would write the binary equivalent of 100_{10} as 110 0100_b.

Adding binary numbers is just like adding decimal numbers except that you carry when you add 1 and 1. for example $10_b + 11_b = 101_b$. When you add the two least significant bits (the 2^0 place) you add 0 + 1 = 1. In the two's column, you add 1 + 1 = 10 but you have to carry the 1 to the next column, the 4's column to get the 101_b. (If it is clear we are talking about binary numbers, I often omit the subscript b.)

One can also represent negative numbers in binary and this is often done using a two's compliment form. I will not discuss this here since we will not be able to spend too much time

on adding and subtracting binary numbers. You might worry more about this in a computer science class.

A four digit decimal number can represent the numbers 0 through 9999, or 0 through $10^4 - 1$. similarly a four bit binary number can represent the numbers 0 through 1111_b, or 0 through $15_{10} = 2^4 - 1$. A sixteen bit binary number can represent the decimal values of 0 through $2^{16} - 1$. It can take a lot more bits than decimals to represent a large number.

In nomenclature, an eight bit number is called a byte, and a four bit number is sometimes called a nibble.

C. Using Logic Gates

A common use of logic gates is to compare two binary numbers. This amounts to recognizing a certain pattern of bits in each number. For instance if you have two binary numbers (A_3,A_2,A_1,A_0) and (B_3,B2,B_1,B_0), how would you test to see if they are equal. You would have a system of gates that will have as inputs the four bits from A, i.e. (A_3,A_2,A_1,A_0) and four bits from B, i.e. (B_3,B2,B_1,B_0) and have one output line, Y. You could represent the system or array of gates as a box with the inputs A and B and the output Y. You might want Y=1 if they are equal and Y=0 if they are not.

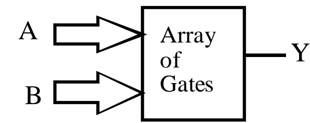

Fig. 9.8

There are a couple of ways of approaching this problem; one using brute force and another involving intelligent guesses. For simple situations, the second often works well, but the first method will always work. The brute force method involves setting up a truth table and implementing it line by line. This often requires a large number of gates and can be cumbersome. I will illustrate both methods for **two bit** numbers, A = (A_1,A_0) and B = (B_1,B_0). The truth table for the situation is shown at the right. In the brute force method you can choose to either implement the cases where Y = 0 or the cases where Y=1. Since there are only four cases where Y=1, it will be easier to do that one. One implements the "ONES" by ANDing the conditions for each line of the truth table that yields a Y=1.

A_1	A_0	B_1	B_0	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

For the first ONE, A_1=A_0=B_1=B_0 =0, or $\overline{A_1}\ AND\ \overline{A_0}\ AND\ \overline{B_1}\ AND\ \overline{B_0} = Y_1$. The gate structure for this is shown at the right. Since this is the first row for which Y =1, I've called the result of this arrangement Y_1. Note that this is a four input AND gate where each input has been inverted, i.e. the bubbles indicate the inversions.

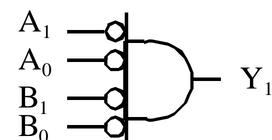

Fig. 9.9

For the second case, Y=1 when $A_1 = B_1 = 0$ AND $A_0 = B_0 = 1$. This can be written logically as $\overline{A_1}\ AND\ A_0\ AND\ \overline{B_1}\ AND\ B_0 = Y_2$, where I've called this result Y_2, since it is the second instance where Y =1. Its circuit is shown at the right. Note that there are no bubbles for the A_0 and B_0 inputs. $Y_2 = 1$ if and only if $A_1 = B_1 = 0$ AND $A_0 = B_0 = 1$. You should be able to figure out how to construct the gates for recognizing the third and fourth patterns that should result in Y=1. Once you have all four cases worked out, you can just OR the results, since that will result in a 1 if and only if at least one of the Y's, Y_1, Y_2, Y_3, or Y_4, is 1. (Note that only one of these Y's can be 1 at a time.) This is accomplished with an OR gate, a 4-input OR gate, with the Y's as the inputs. This would require a total of 8 INVERT gates, 4 AND gates and one OR gate.

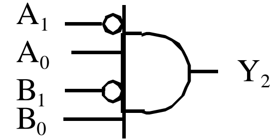

Fig. 9.10

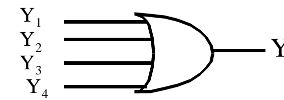

Fig. 9.11

The other method in solving this problem would be to recognize that for the two numbers to be equal, the two bitwise comparisons, i.e. A_0 to B_0 and A_1 to B_1, must each be NOT exclusive OR, or NOT EXOR, or be EXNOR. This result can then be ANDed to get the final Y output. See the gate arrangement at the right. The output of this arrangement of gates, Y, is 1 if and only if A=B. To extend this to compare two, four bit numbers, you would need two more EXNOR gates and a four input AND gate instead of a two input AND gate.

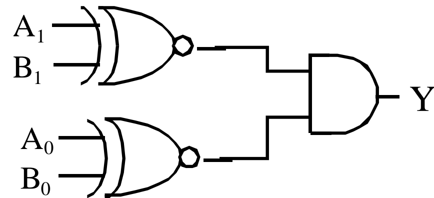

Fig. 9.12

A slightly more complicated situation is to try and decide is A > B or if A =B. Here there are actually three possibilities, A < B, A=B or A > B. To represent three different possibilities, you will need an output the has two digital lines, or a two bit number. With a one bit number you can only represent two possibilities, 0 and 1. With a two bit number you have four possibilities, 00, 01, 10, and 11. If you just wanted to know if A ≥ B, you would only need a one bit number, say 1 for A ≥ B and 0 for A < B. However, to distinguish A > B from A = B you need two bits.

For A = B the above circuitry will detect that. If A > B it is a little more complicated. For a two bit number there are just two possibilities, ($A_1 = 1$ AND $B_1 = 0$) OR {$A_1 = B_1$ AND ($A_0 = 1$ AND $B_0 = 0$)}. The first condition is just the gate arrangement on the left below and the second possibility is the middle one. If you OR Y_1 and Y_2 together you have the desired solution

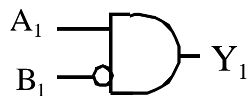

Fig. 9.13

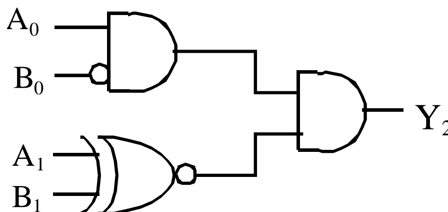

Fig. 9.14

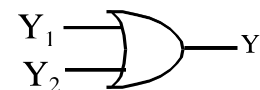

Fig. 9.15

of Y = 1 of A>B and 0 if it's not greater than B. It gets more complicated when you compare four bit numbers, but this should give you an idea of how the gates are used. I have copied the gate configuration for the 7485 four bit magnitude comparator from the Texas Instruments TTL Data Book so you can see how they do it. They usually try to minimize the number of gates they

use to implement the function(s), so it can be hard to see just what they are doing. Note that there are often many ways to implement a given function. The first example above gives an indication of that. If you are making a magnitude comparator, or any other device, it is usually financially advantageous to minimize the number of gates (i.e. transistors) used. The inputs are on the left side and the outputs are on the right. It is useful to note that when two lines cross, they are only connected if there is a solid dot there. The open dots indicate invert operations. You should also note that the output consists of three bits, one for A=B, one for A<B and a third for A>B. This makes it easier to cascade two four bit comparators to make and eight bit comparator. Also the three inputs (2), (3), and (4) or A<B, A=B and A>B are for cascading the outputs from another comparator to make an eight bit comparison.

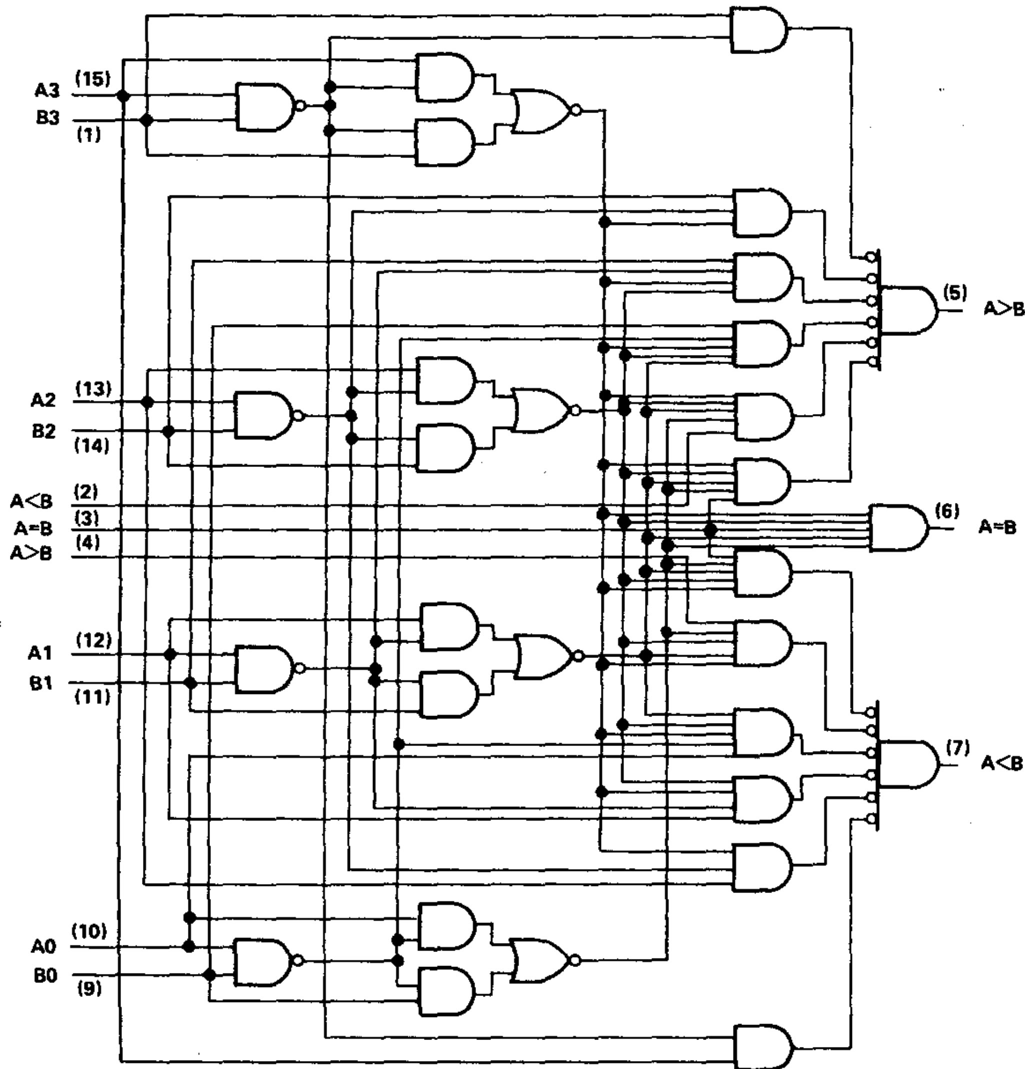

Fig. 9.16 A 7485 Magnitude Comparator

I should point out that this is fairly old technology, the copyright in the data book is 1976, but the logic is still the same now as it was 30 or 40 years ago. What I want you to realize is that you can use these gates to perform some surprisingly complicated logical and mathematical operations. However, you may have to use a lot of gates.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The last example is a binary adder. If you have two one bit binary numbers, A and B, and want to add them, you want to implement the truth table at the right. Note that the "sum" has two bits, or one more bit than the original numbers, since there can be a carry if both A and B are 1. In the table, Sum is the LSB of the addition and the Carry is the carry out if both A and B are 1. The sum part of the table can be implemented by an EXOR gate, and the Carry by an AND gate. The circuit is shown at the right and this circuit is called a half adder. The half adder is enclosed in the box with a dashed line. A full adder would have to allow for a carry in from the addition of a less significant bit pair, i.e. if you had two bit numbers, this circuit would suffice to add the least significant bits, A_o and B_o, but you would have to have some provision or a carry from this addition to the addition of A_1 and B_1. Such a device would be called a full adder. A full adder can be made from two half adders and an OR gate. Here there are potentially three inputs, A and B and a potential carry in from the previous less significant bit's addition. The circuit is shown below.

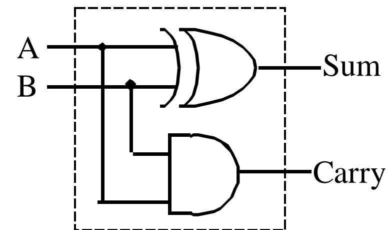

Fig. 9.17

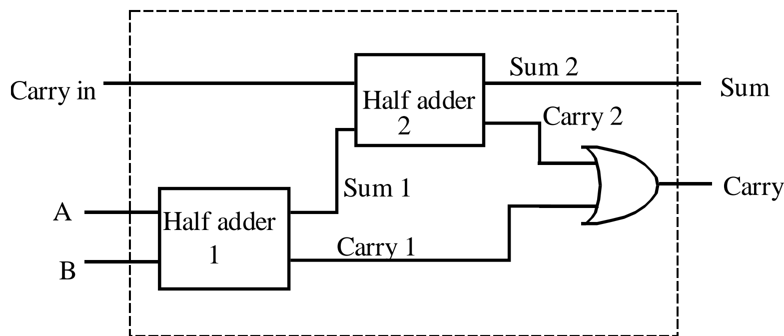

Fig. 9.18 Full adder

I usually don't ask you to figure out how to perform complicated logical operations with gates, but rather see if you can figure out the truth tables for a given arrangement of gates.

D. Electrical Characteristics of the TTL Logic Family

There are several ways of making gates out of transistors. There are two we will consider. The first is TTL or Transistor – Transistor Logic. Within this major family, there are several subtypes. The simplest NAND gate, e.g. a single gate in a standard 7400 requires four transistors and several diodes. (A 7400 is an integrated circuit containing four 2-input NAND gates.) The standard TTL gates have were largely replaced by the LSTTL family (LS = Low power Schottky) which was replaced by the ALSTTL or Advanced LSTTL. There are some other

subfamilies which I will not mention. In the 90's these tended to be replaced by CMOS technology that uses MOS transistors rather than the bipolar junction transistors of the TTL family. In the beginning the TTL families were faster, but used more power per gate than the MOS family. Gradually the speed of the MOS transistors improved and they became very fast. Another important factor is that it is much easier to make very large scale integrated circuits (millions of transistors per chip) with MOS transistors. However, the MOS transistors are more sensitive to static discharge than the bipolar transistors and as they switch more rapidly, their power consumption increases. This increase in power consumption can be partially offset by reducing the supply voltages.

For **TTL logic gates**, an input voltage between 0V and 0.8V is guaranteed to be recognized as a Lo state or 0, and a voltage between 2V and 5V is guaranteed to be recognized as a Hi state or a 1. At the output, a Lo output from a TTL gate is guaranteed (under "normal" conditions) to be < 0.4V and the Hi output is guaranteed to be > 2.4V. Note that this implies that the output from a gate is going to be at least 0.4V below the value that is guaranteed to be recognized as Lo by the input to another gate. We would say that there is 0.4V of noise immunity. Similarly the output Hi from one gate is guaranteed to be 0.4V > than the level guaranteed to be recognized as Hi at the input of another gate, again giving 0.4V of noise immunity. Typically the input voltage that produces a transition from Lo to Hi at the output of the gate is between 1.0V and 1.4V for these devices.

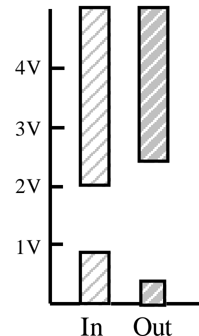

Fig 9.19 TTL Logic levels

The power supply voltages are 0V for the common or ground and +5V for the high power supply voltage, usually labeled V_{CC}. Do not let the positive power supply connection get higher than about 5.5V or less than 4.5V. Usually you try to have $V_{CC} = 5.0V \pm 5$ %.)

Conceptually one <u>might</u> make a TTL inverter as shown below, but that is not how they are made. This is just a transistor in a common emitter configuration. If the input is less than about 1.2V, the transistor is off and $V_{out} = 5V$, a High state or 1. If the input is much above 1.2V the transistor is on and the output is close to 0V, a Low state or 0. R_B is necessary to limit the base current and $R_B \approx R_C \approx 2.2k$ to insure a high gain so that a small increase in V_{in} beyond 1.2V will make the output go from about 5V to almost 0V. The diode is necessary to make sure a 0.8V input will not turn the transistor on and drive the output Lo. A "real" TTL inverter would not be made this way because this configuration is not optimal and may not switch as fast as desired.

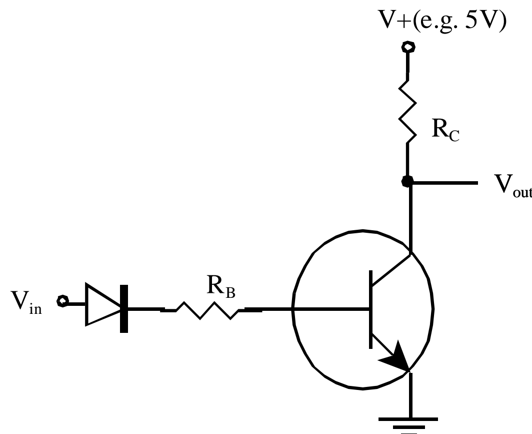

Fig. 9.20 An Inverter

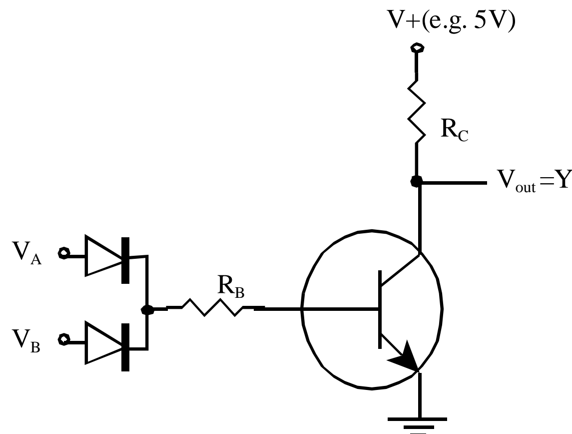

Fig. 9.21 A NOR gate

One could easily convert the above inverter (fig. 9.20) into a NOR gate by adding a second input as shown in fig. 9.21 above. Again, this is not how it is actually done, but it gives you an idea of how it could be done. Sometimes you actually do something like these if you only need one gate and do not want to add a TTL integrated circuit to the system. It is sometimes called Mickey Mouse Logic.

If one of the inputs to a true TTL gate is unconnected, the gate will interpret the input as a Hi state. To take the input Lo, one must actually pull current out of the gate. (There is an internal resistor that tends to pull the input Hi.) If you want the input to be Lo you have to ground it, or connect it to ground via a relatively small resistor, $R < 500\Omega$ for standard TTL. (It is best to connect it directly to the power supply ground though.) If you want the input to be Hi, you usually connect it to the +5V supply through a resistor, e.g. $R \approx 4.7k\Omega$, or by connecting it to the output of another gate whose output is Hi.

The TTL family of logic gates was packaged in a series integrated circuits in standard configurations. The devices were given a code 74xxx, where the series was designated by the leading 74 and the particular integrated circuit was specified by the numbers xxx. The device 7400 was an integrated circuit containing four dual input NAND gates. The 7402 contains four dual input NOR gates. There were many different configurations and functions implemented in this 74xxx series. A 7400 is a standard TTL device, the 74LS00 is the same configuration but using low power Schottky technology. The 74ALS00 uses advanced low power Shotkey technology.

D. Electrical Characteristics of the CMOS Logic Family

The CMOS logic family also has some subfamilies. The original CMOS gates could operate on a wide range of power supply voltages. They were usually specified to operate with $5V < V_{CC} < 15V$. Packages with this technology were usually labeled CD4xxx. For example, a CD4001 is an integrated circuit that contains four dual input NAND gates. However, the pin configuration is not the same as the 7400. Later versions of the CMOS devices were packaged with (in most cases) the same pin configuration and they were labeled 74Cxxx, the C referring to CMOS. A 74C00 was a package of four dual input NAND gates with the same pin out as the 7400. However, they were not entirely compatible with the TTL series. For instance, when operating from a 5.0V supply, the input Lo voltage was guaranteed to be recognized as a Lo if $V < 1.0V$. The input was guaranteed to be recognized as a Hi if $V > 4.0V$. Similarly the Hi output of a gate is guaranteed to be >4.5V and the Lo output is < 0.05V, providing about 0.5V of guaranteed noise immunity. Typically the actual transitions levels occur around 2.5V at the input. Also, the output cannot sink very much current and they may not be able to pull the input of a TTL gate low. (They will usually pull an LSTTL gate low.) As a result you avoid mixing TTL components and the old style CMOS components in the same circuit.

Newer CMOS families are compatible with the TTL families and you should try to use those and avoid the older CMOS unless you have a special reason for using them. The newer families are labeled 74HCTxxx or 74ACTxxx. (The 74HCxxx and 74ACxxx are 'sort of' compatible, but not guaranteed compatible, with the TTL devices. Use the HCT or ACT series when mixing CMOS and TTL components in a circuit.)

CMOS inputs require no current in the static state, only when switching. This is because the input really acts like a capacitance. Once it is charged, you do not need a current to maintain that voltage; you only have a current flow when the voltage is changing. (There are leakage currents, but they are usually small, typically $<< 1\mu A$.)

There are also CMOS families designed to operate at lower voltages. Operating at lower voltages reduces power consumption and heating compared to when they operate at higher voltages. The lower voltages also help reduce the magnitude of the electric fields in devices made with smaller geometries. That reduces the likelihood of other problems like dielectric breakdown in the MOSFET's gate. Most CPU chips operate on lower voltages. I will not discuss the characteristics of these devices.

E. Pratical Considerations

Now we typically use these circuits as "glue" to interface experiments to computers or other devices in "one of a kind" situations. If a device is mass produced they will often integrate many of these functions into larger integrated circuits, so the use of individual IC's in the 7400 series is much less extensive than it was twenty-five years ago. Nevertheless it is useful to know how these devices work, since they are the functional building blocks of these larger integrated circuits and understanding these building blocks will help you understand how the larger circuits work. From an experimental point of view, understanding how a device works will help you know how and when to use it and help you use it correctly.

F. CMOS Gates

I have included some figures of CMOS gates. The bubbles on the gate connections indicate that the transistor is a P channel MOSFET.