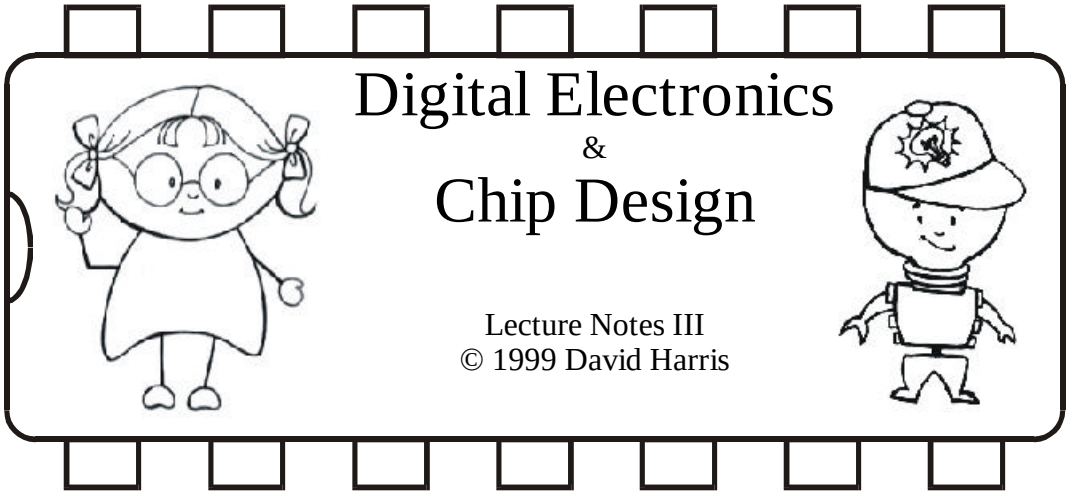


Now that we have learned the basics of digital electronics, we will look at several topics of interest for building larger, more complex and practical digital systems. In particular, we will study Boolean algebra somewhat more formally, we will learn about a canonical form for Boolean expressions, and we will study representing numbers in the binary and hexadecimal systems. We will conclude with techniques for breadboarding projects.

Boolean Algebra

Just as with real variables, Boolean variables obey a variety of properties that let one simplify or rearrange expressions. Order of operations, of course, is crucial to interpreting Boolean expressions. As multiplication takes precedence over addition, the Boolean equivalent AND takes precedence over OR. A bar over a Boolean expression indicates to compute the expression, then take the complement (do a NOT operation) on the result. Below are a list of useful properties:

$\overline{\overline{A}} = A$	
$(A + B) + C = A + (B + C)$	Associative Property
$A + B = B + A$	Commutative Property
$A + 0 = A$	
$A + 1 = 1$	
$A + \overline{A} = 1$	
$(A \bullet B) \bullet C = A \bullet (B \bullet C)$	Associative Property
$A \bullet B = B \bullet A$	Commutative Property
$A \bullet 0 = 0$	
$A \bullet 1 = A$	
$A \bullet \overline{A} = 0$	
$(A + B) \bullet C = A \bullet C + B \bullet C$	Distributive Property
$(A \bullet B) + C = (A + C) \bullet (B + C)$	Distributive Property
$\overline{A + B} = \overline{A} \bullet \overline{B}$	DeMorgan's Law
$\overline{A \bullet B} = \overline{A} + \overline{B}$	DeMorgan's Law

The last two rules are probably the only non-obvious ones. DeMorgan's Law is frequently useful because it allows the conversion of ANDs to ORs and vice versa using appropriately placed inversion.

Let us apply these rules to simplify a hideous expression:

$$\overline{(\overline{\overline{X}} + \overline{\overline{Y}}) \bullet [(A \bullet B) + (\overline{\overline{A}} + \overline{\overline{B}})]}$$

$$= \overline{(\overline{\overline{X}} + \overline{\overline{Y}}) \bullet [(A \bullet B) + \overline{(\overline{A} \bullet \overline{B})}]}$$

$$= \overline{(\overline{\overline{X}} + \overline{\overline{Y}}) \bullet 1}$$

$$= \overline{(\overline{\overline{X}} + \overline{\overline{Y}})}$$

$$= X \bullet Y$$

Canonical Form

As we have just seen, there are many games one can play with Boolean expressions. However, just as the digital abstraction traded away the full range of analog voltages for a simpler system that is easier to work with and thus has more practical power, we sometimes prefer to trade away the full range of Boolean expressions for a standard notation, or so-called "Canonical[1] Form." An ideal canonical form would still be able to represent all possible Boolean functions. However, it would be easy to implement with efficient circuits and easy for computers to process in automatic logic generation programs. The "Sum of Products" canonical form fits all of these criteria.

In the sum of products form, all expressions are made from terms consisting of variables and the complements (NOTs) of variables ANDed together. These terms are then ORed together to give the expression.

For example, observe how the following expression is transformed into canonical form:

$$\overline{A + B} + C \bullet (\overline{D} + E)$$

$$= \overline{A}\overline{B} + C \bullet (\overline{D} + E)$$

$$= \overline{A}\overline{B} + C\overline{D} + CE$$

Since ANDs are written as multiplication and ORs are written as addition, the resulting expression is justly called sum of products. Note that we frequently omit the multiplication sign within the product terms.

[1]Canonical is just a fancy word that just means standard. It was probably brought into use by scientists and engineers who wished to confuse laymen with fancy jargon.

Now that we have the expression in canonical form, we can implement it with NOT gates, several multiple-input AND gates, and a single multiple-input OR gate. For instance, in this case, we need three NOT gates, three two-input AND gates, and a three-input OR gate. If we are lacking the proper multiple-input gates, we can always use a cascade of two-input gates instead.

Binary and Hexadecimal Numbers

Humans use base 10 with the digits 0-9 to represent numbers. Unfortunately, this system is not efficient for digital systems. The smallest unit of information, the bit, is either a 0 or a 1. Thus, we like to use the binary system to represent numbers using only 0's and 1's. Instead of having the 1's column, 10's column, 100's column, 1000's column, and so forth as we are accustomed, the binary system uses the 1's column, 2's column, 4's column, 8's column, etc. A subscript 2 is sometimes used to indicate that a number is in binary form when it is not obvious from context. For instance, the binary number 101010_2 = 0*1 + 1*2 + 0*4 + 1 * 8 + 0 *16 + 1 * 32 = 42 in the decimal system. Multiplication and addition are performed as one might expect:

```
 1001        1001
+0101       x0101
-----       -----
 1110        1001
           1001
           ------
           101101
```

While the binary system is excellent at the level of digital logic, it is rather cumbersome for humans computation because one must keep track of a large number of digits. Thus, especially when working with computers, we also use the hexadecimal system, also known as base 16. Four binary digits, representing 2^4=16 possible values, are lumped into a single hexadecimal digit. We need 16 symbols to represent the digit; we use 0-9, then overflow to use A-F. The table below shows the correspondence between binary, hexadecimal, and decimal numbers:

Binary	**Hexadecimal**	**Decimal**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10

1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

To convert from binary to hexadecimal, we group the binary digits into groups of four, padding with 0's on the left when necessary. From the earlier example, 101010_2 is grouped as 0010 1010. Now, we just read off the hexadecimal digits from the table to get the base 16 representation 2A. Often, to denote that a number is in hexadecimal form, we write it with a $ sign: $\$2A = 101010_2 = 42$.

We call the group of four binary bits a "nibble." In addition to conveniently translating to hexadecimal notation, a nibble is frequently encountered in digital logic chips. For instance, the 7483 adder chip adds two four bit numbers. By cascading several such adders, we can add an arbitrarily large binary number.

Beyond nibbles, we call eight bits a "byte." A byte can represent a number from 0 to 2^8-1 = 255. The largest commonly used term is a "word." It is typically the number of bits used by a microprocessor. Earlier processors had 8 and 16 bit words. 32 bits is now a typical word length, but some special purpose processors use words of 36 bits or more. The latest RISC (Reduced Instruction Set Computer) processors are even using 64 bit words.

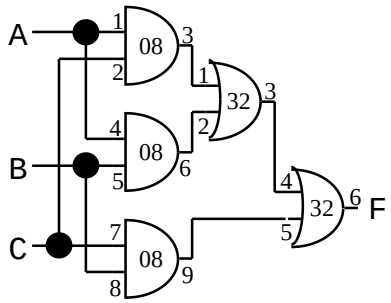
Breadboarding Techniques

Moving from theory to application, we will study a few techniques that will make breadboarding easier. In particular, most novice digital designers find that debugging is their greatest difficulty. Many of these techniques are intended to reduce the number of problems introduced into circuits. Good engineers recognize that they will likely make virtually every possible stupid mistake. Therefore, the secret to building bug-free circuits is not superior intelligence, but rather to learn design habits that minimize the likelihood of making mistakes and the damage done if mistakes are made.

The first rule of circuit construction is to keep the power off while building a circuit. TTL logic gates are fairly robust by now and are seldom damaged, but finding defective chips in a circuit is always difficult. Moreover, keeping power off is an important habit to develop before working with more sensitive chips.

The second rule is to be sure you have turned on the power when you are ready to operate the circuit. Make sure the power LED on your board turns on. Though this seems obvious, this author has probably spent more time scratching his head because he forgot to turn on the power than for any other single bug. If power is on but the power LED is not on, you probably have a short circuit between power and ground. Your transformer may be getting very hot. Turn off the power immediately and check for shorts.

When first drawing your design, label each of your gates on paper with the chip that it is actually on. Then label each input and output on your schematic with the actual terminal used on the chip. Then when you proceed to build the circuit, you can simply wire from point to point as shown on your schematics. Be very careful with the 7402 NOR gate. While most gates have their inputs at pins 1 and 2 and the outputs at pin 3, the 7402 is backward, a frequent source of error. For example, one might label the example circuit from the previous course notes as follows:



When your schematic is labeled, place your chips on the breadboard. Learning to read electronic component part numbers can be an art in itself. The TTL logic gates come in many families: original, S, LS, ALS, HC, and more. Each chip has information printed on it, usually listing the manufacturer, the part number, and perhaps some other information that generally serves to confuse the user. One line should read *74LSxx*, where the first * indicates the manufacturer (perhaps M for Motorola, SN for Texas Instruments, or DM for National Semiconductor), the 74xx is the component number (for example, 7408 for an AND gate), LS stands for Low Power Schottky (an improved technology over the original TTL gates featuring greater speed and lower power consumption), and the final * indicates the physical packaging style (generally P or N). For this class, we will be using LS technology chips. The only information you need to correctly select your chip is the 74xx label.

If possible, place gates that are used together physically close to each other to reduce the amount of wire necessary. Before doing anything else, connect power and ground to the appropriate terminals of the chips. Forgetting to do so is another very common mistake.

Finally, when wiring from one chip to another, don't run wires directly over a chip. If, for some reason, you have to remove the chip (perhaps it is defective or it is the incorrect chip), you should be able to do it without disturbing any wires already placed. Also, wire neatly. A neat circuit is much easier to trace and debug than a sloppy one and you will almost always find that the time required to wire neatly is more than compensated for by the time saved in chasing errors.

Debugging is largely a matter of experience. If you are not getting the expected output, go back to the very beginning of the circuit. Use a wire controlling an LED to probe voltages. Check that you have the proper inputs. Trace the logic through and check that each gate has the proper output. In this manner, you can quickly track down many bugs caused by incorrect wiring or defective chips. Ultimately, the only way to become good is

to build and debug many circuits. For that reason, you will have a great deal of practice constructing circuits in this class.