

Under the Hood III: Digital Logic

The logic of Aristotle underlies the modern digital computer. Logical rules developed to analyze the truth of verbal arguments led eventually to symbolic logic where true and false become 1 and 0, respectively, following the work of Claude Shannon in the last century. Symbolic (or propositional) logic can be used to formulate algebraic operations. The final steps, exemplified by the electronic computer, were to convert true and false values to voltages and logical operations to simple circuits, simple circuits that can be combined in the millions to produce a computer.

Creating real computers out of logic units is a field in itself but we will try in Part IV to outline the steps, wave our hands, and describe enough to convince you of the connections. In the process, you'll design some simple circuits. Here we define four "connectives," AND, OR, XOR, and NOT, that allow us to create new "compound" statements from existing statements. Of course, once statements are combined with connectives, the results are themselves statements and can be further combined to yield a rich structure. The study of elementary mathematical logic derives largely from the way these connectives are defined.

Logic

Consider the four statements below:

1. the sun is hot
2. the moon is made of green cheese
3. diamond is a hard material
4. gold ingots float on water

Experience tells us that the first and third are true statements while the second and fourth are false. Let's assume there are no tricks here and the statements really are true and false as stated.

AND

What we are interested in is the truth or falsity of combinations of the statements, as in:

- the sun is hot AND diamond is a hard material
- the sun is hot AND the moon is made of green cheese
- the moon is made of green cheese AND diamond is a hard material
- the moon is made of green cheese AND gold ingots float on water

Our criterion for truth is that the statement as a whole must be true. With this idea, we would say that the first statement is true but all of the rest are false.

OR

Now look at the OR operation:

- the sun is hot OR diamond is a hard material
- the sun is hot OR the moon is made of green cheese
- the moon is made of green cheese OR diamond is a hard material
- the moon is made of green cheese OR gold ingots float on water

In this case, all of the statements except the last would be considered to be true. If one part or the other of the statement is true, we can claim that the statement as a whole is true.

XOR

You might actually have a quibble with the previous statement if you think that OR should be restricted to EITHER...OR. In this case, you would consider the first statement to be false also since it is not the case that EITHER the sun is hot OR diamond is a hard material. Actually, this form of logical combination goes by the name of an exclusive OR, also called XOR, while the "regular" OR is an inclusive OR.

- the sun is hot XOR diamond is a hard material (we take this to be FALSE)
- the sun is hot XOR the moon is made of green cheese (TRUE)
- the moon is made of green cheese XOR diamond is a hard material (TRUE)
- the moon is made of green cheese XOR gold ingots float on water (FALSE)

NOT

The simplest logical operation is just negation (sometimes called "denial"), if a statement is true then its negative is false and vice-versa.

- the sun is NOT hot
- the moon is NOT made of green cheese

Since the sun is hot was a true statement the sun is NOT hot is false.

The operations described above are at the heart of the design and construction of the chips and circuit boards of a digital computer. Each basic operation shown here is implemented in tiny circuits called logic "gates" (nothing to do with the well-known chief of Microsoft Corporation). So in building a chip, we might have a collection of AND gates, OR gates, NOT gates, etc, that go together on a board to implement the functions that we might want to achieve. We'll talk more about these as we go along.

Digital Logic

AND, OR, XOR, NOT

The logical results of the operations can be summarized in so-called truth tables. Here is the truth table for AND (sometimes written as $X \wedge Y$ or as an implicit multiplication, $X*Y$, or just XY)

AND

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

Next for the OR operator, written as $X \vee Y$ or often using an addition-like symbol $X + Y$.

OR

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

The exclusive OR, called XOR, symbolically written $X \oplus Y$,

XOR

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

And finally, negation, written as $\sim X$ or X' .

NOT

X	$\sim X$
0	1
1	0

Having defined these operations, we might note that the rows of a 2-variable truth table can be denoted by the compound values that they represent. For instance, we can represent the rows as in the following table (Figure 8-28):

X	Y	Compound representation	Same using primes
0	0	$(\sim X)(\sim Y)$	$X'Y'$
0	1	$(\sim X) Y$	$X'Y$
1	0	$X(\sim Y)$	XY'
1	1	XY	XY

Figure 8-1 Minterms corresponding to XY Boolean values

Exercise: Set up a table like this one for three variables, X, Y, and Z.

Gregg, 1998, has a good discussion of all of the possible truth tables that can arise from combining X and Y in some fashion. If you think of a particular truth table for an operation as just 4 bits, then it is clear that there are 16 possible outcomes. The table shows all of them, with a few of the operations given names. For convenience, we number the columns for the 16 outcomes.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
X	Y	F	NOR		$\sim X$	>	$\sim Y$	XOR	NAND	AND	iff	Y	Implication	X		OR	T
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 8-2 Possible outcome combinations

We will discuss below the choice of headings for the columns that we haven't dealt with yet.

Notice that Column 1 of Figure 8-29 corresponds to the totally FALSE statement, sometimes denoted "0" -- there is a false result for any combination of truth values for the components, X and Y. Similarly, Column 16 is the statement that is always TRUE or "1".

IMPLICATION (Column 12)

This is the truth table for the idea: "X implies Y," written $X \rightarrow Y$, or sometimes "if X then Y." A good example of that is Pavlovian conditioning. Consider the statement:

If a bell rings, the dog gets food.

Bell Food		"Reinforcement" Bell \rightarrow Food	
0	0	1	reinforces dog's belief (no bell, no food)
0	1	1	doesn't hurt, dog won't complain
1	0	0	fight against reinforcement
1	1	1	reinforces dog's belief

Another useful example for understanding implication is to think of the statements:

X: It rains on Saturday.

Y: I go to the movies.

Then $X \rightarrow Y$ is the statement: If it rains on Saturday, then I go to the movies. In this case, we ask in which cases have we been untruthful? In the first two cases, it didn't rain and we either did not or did go to the movies. But we had not told a lie! It's the third case (it rained and we didn't go to the movies) where we definitely have a false statement -- value 0. Finally it rains and we go to the movies -- perfectly fine.

Rain	Movies	Rain \rightarrow Movies	
0	0	1	True: no lie has been told
0	1	1	True: no lie has been told
1	0	0	False: we told a lie
1	1	1	True (clearly)

An interesting sidelight here is that this compound statement $X \rightarrow Y$, could also be expressed as $(\sim X) \vee Y$ or, in other notation, " $\sim X + Y$." So we could construct an implication "gate" by taking a $(\sim X)$ gate and adding that to a Y gate. So we could just as well have headed Column 12 of Figure 8-29 with " $(\sim X) + Y$."

A note on notation: Without extra parentheses, the "NOT" operation is always carried out BEFORE either the "AND" or "OR" operations. This means, for example, that expressions like " $(\sim X) + Y$ " can unambiguously be written " $\sim X + Y$." If the denial is on the second symbol, parentheses are preferred: " $X + (\sim Y)$ " rather than " $X + \sim Y$ ". If, on the other hand, we want to have an AND or an OR carried out BEFORE a denial, we write this using the parentheses: " $\sim(X + Y)$," or " $\sim(XY)$." The situation is the same with the "prime" notation for negation: $X + Y' = X + (\sim Y)$, and $(X + Y)' = \sim(X + Y)$. For AND, the same principles apply: $XY' = X * (\sim Y)$ whereas with parentheses the AND is carried out first $(XY)' = \sim(XY)$. We shall use these notational conventions in the future.

GREATER THAN (Column 5)

Looking at the fifth column of the full chart, we see that its truth values are precisely the opposite from those in Column 12. We denote Column 3 as ">" since if the values of X and Y were taken from the real numbers 0 and 1 (rather than FALSE and TRUE), then the statement " $X > Y$ " would be false except in the case where $X = 1$ and $Y = 0$. Notice also that the truth values in this column are exactly the opposite of what is labeled "Implication" in Column 12 (Figure 8-29). So in every case, this is the denial of $X \rightarrow Y$. This column, again, could have been headed in any of these equivalent ways: $\sim X \wedge Y$, $\sim X * Y$, $\sim XY$, or $X'Y$.

IF AND ONLY IF (IFF - Column 10)

This is true if X and Y have exactly the same truth values. We denote it "X iff Y" or " $X \leftrightarrow Y$ " or merely " $X = Y$ " or, writing it out as "X if and only Y"

X	Y	X = Y
0	0	1
0	1	0
1	0	0
1	1	1

NAND (NOT AND, Column 8)

This gate, " $\sim(XY)$ " is an important one for two reasons. One is that it is easily implemented with transistors and the other is that it can be thought of as a "universal gate" since all other logic gates may be constructed from it. In the same way, the NOR gate " $\sim(X + Y)$ ", (Column 2) is a universal gate.

Exercise: How would you describe Columns 3 and 14 using the operations of AND, OR, and NOT?

Multi-way AND and OR

The truth table for an AND statement with more than two inputs can be reasoned out in a couple of ways. In a statement such as **if $A * B * C$** , one can work out a truth table for combining **$A * B$** , then use the result to combine with **C** ; or one can combine **A** with the result of the statement **$B * C$** . Look at the following table.

A	B	C	A*B	(A*B)*C	B*C	A*(B*C)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Figure 8-3 Associativity of AND

Notice, significantly, that the two columns indicated with an asterisk are exactly the same. Hence, we have that AND, as an operation, is what the mathematicians call "associative". That is:

$$(A*B)*C = A*(B*C).$$

Notice also that either of these expressions is TRUE only in the situation where **all three** of A, B, C are TRUE. This leads us to a meaningful way to **define** the expression without parentheses, $A*B*C$, namely, if all three of the constituent parts of the expression are true. Exactly the same kind of an argument can show us how to define the expression $A+B+C$.

Exercise: Go through reasoning similar to what we have just done for defining $A+B+C$.

Building components from AND, OR, and NOT.

Here is how one can build an XOR gate from NOT, AND, and OR. From Column 7 of Figure 8-29, that **$X \text{ XOR } Y$** is true (has value **1**) if the 2nd row is true or if the 3rd row is true; that is if **$X*Y'$** is true OR if **$X'*Y$** is true. Hence, the XOR function can be written as

$$F = X * Y' + X' * Y \quad \text{or we say simply } F = X Y' + X' Y$$

Where the function means that the statement is true if (X is true **and** Y is not) **or** (Y is true **and** X is not). Typically the * will not be written but will be implied as in the right hand expression, just as is the convention of the use of concatenation for multiplication in elementary algebraic statements.

A light approach to Boolean Algebras

The structure we have been looking at, a symbolic logic system, represents a special case (having two values, TRUE and FALSE) of the general entity, the Boolean algebra. Mathematics loves abstraction. Mathematics loves to frame the development of a system as a set of definitions and axioms which lead to results that will hold for any situation for which the mathematical statements hold. So, for instance, the celebrated Boolean algebra has many examples (logic, set theory, the so-called distributive and complemented lattices, etc.), and so a single theorem in Boolean algebra theory is a theorem in each of its examples. There is great power in this sort of thing! First the definition.

Definition: A **Boolean algebra** (BA) is a system consisting of a nonempty set S , two binary operations ($+$ and $*$), a unary operation ($'$), and two special distinct elements, 0 and 1 , satisfying the following axioms for all elements a, b , and c in S [we drop the $*$ in this definition except where it contributes to clarity]:

B1. Commutative Laws:	$a + b = b + a$	$ab = ba$
B2. Associative Laws:	$a + (b + c) = (a + b) + c$	$a(bc) = (ab)c$
B3. Identity Laws:	$a + 0 = a$	$a * 1 = a$
B4: Complement Laws:	$a + a' = 1$	$aa' = 0$
B5: Distributive Laws:	$a + bc = (a + b)(a + c)$	$a(b + c) = ab + ac$

Often the BA is taken to be the system $(S, +, *, ', 0, 1)$ where the objects in this six-tuple have the properties shown in the definition.

People generally have no problem imagining that these axioms are valid for logical and set-theoretic systems until they get to the first of the distributive laws (which combine the $+$ and $*$ operations). You can demonstrate this easily for 2-valued logics with a truth table.

A	B	C	B*C	A+B*C	A+B	A+C	(A+B)(A+C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

The values in the two starred columns are identical demonstrating that $A+B*C = (A+B)(A+C)$ holds in the truth-table formulation of propositional logic. In fact, any of the axioms of Boolean algebra can be shown to hold for the logic we have been studying where + is OR, * is AND, and ' is NOT. That is, our logic of statements forms a Boolean algebra.

Exercise: Use truth tables to show that the statements in B4 and the right-hand statement in B5 hold for propositional logic, as we have done in Figure 8-31 with the one distributive law.

Once we have shown that our logic is a BA, then we can use any of the axioms of the system to do calculations on compound expressions. Suppose that we are given an expression like:

$$F = a'b'c' + a'bc' + ab'c' + abc'$$

and that we wished to simplify the right hand side. Here's how we can apply the axioms to do so:

$$\begin{aligned}
 F &= a'b'c' + a'bc' + ab'c' + abc' \\
 &= a'b'c' + ab'c' + a'bc' + abc' && \text{[using B1]} \\
 &= (a' + a)b'c' + (a' + a)bc' && \text{[using B5]} \\
 &= 1 * b'c' + 1 * bc' && \text{[using B4]} \\
 &= b'c' + bc' && \text{[using B3]} \\
 &= (b' + b)c' && \text{[using B5 again]} \\
 &= 1 * c' && \text{[using B4 again]} \\
 &= c' && \text{[using B3 again]}
 \end{aligned}$$

So now we have that $F = c'$, a substantial simplification from the original expression for F.

Some important theorems about Boolean Algebras

Here we shall use the definition of a BA $(S, +, *, ', 0, 1)$ to show a few key theorems about general elements from S.

Theorem 1 (Idempotent Laws): For all a in S, $a*a = a$ and $a + a = a$.

Proof: We'll prove the first statement leaving the second for you to do.

$$a*a = a*a + 0 = a*a + a*a' = a*(a + a') = a*1 = a \text{ [using B3, B4, B5, B4, B3, respectively]}$$

Exercise: Show that $a + a = a$.

Theorem 2 (Uniqueness of a') : If $a \in S$, then a' is the only element satisfying $a + a' = 1$, $a*a' = 0$.

[Note. This is one of a class of results that we call "uniqueness" theorems. It says that for a given a , there is one and only one element, namely a' , that satisfies property B4 from the definition of a Boolean algebra.]

Proof: Let's suppose that there exists another element $t \in S$, such that $a+t = 1$, $a*t = 0$. That is, suppose that another element satisfies the same property that a' does. We will show that necessarily this forces t to equal a' .

Now, $t = t + 0 = t + a*a' = (t+a)*(t+a') = 1*(t+a') = t+a'$. Also, by the same kind of reasoning, $a' = a' + 0 = a' + a*t = (a'+a)*(a'+t) = 1*(a'+t) = a'+t = t+a'$. Hence, $t = a'$ and uniqueness holds.

Exercise: Determine the reasons for each of the steps in the theorem's proof above (using the notations from our definition of Boolean algebra, e.g., B1, B2,

Theorem 3 (Idempotent Laws): For every $a \in S$, $a * a = a$, and $a + a = a$.

Proof: Let $a \in S$, and look at

$$a * a = a * a + 0 = a * a + a * a' = a * (a + a') = a * 1 = a$$

Exercises: Supply reasons for each step above. Also prove the other part of the theorem, that $a + a = a$, showing a reason for each step that you use.

Theorem 4 (Domination Laws): For each $a \in S$, $a + 1 = 1$, $a * 0 = 0$.

Proof: [Though this theorem seems strange (first because the first part doesn't sound reasonable, and because the second part seems as if it should be obvious) both parts do hold but need to have a proof before we can use them.]

$$a + 1 = a + (a + a') = (a + a) + a' = a + a' = 1 \quad [\text{B4, B2, idempotence, and B4 again}]$$

Exercise: By similar argument, show the second part of the theorem.

Theorem 5 (Absorption Laws): For each $a, b \in S$, $(a + b)*a = a$, $(ab) + a = a$.

Proof: $(a + b)a = (a + b)(a + 0) = a + b*0 = a + 0 = a$ by B3, B5, Domination law, B3.

Exercise: By similar argument, show the second part of the theorem.

Theorem 6 (DeMorgan Laws): For each $a, b \in S$,

$$(i) (a + b)' = a' b',$$

$$(ii) (ab)' = a' + b'.$$

Proof: We'll use the powerful result from Theorem 2 (Uniqueness of complements). That is, we'll show that $a' b'$ serves as a complement for $a+b$. But by B4, $(a+b)'$ is also a complement of $a + b$, so the first part of the theorem will hold. That's the plan. Now here goes:

$$(i) \text{ Show } (a + b) + a' b' = 1, \text{ and } (a + b)*(a' b') = 0:$$

$$(a + b) + a' b' = a + (b + a' b') = a + (b + a')(b + b') = a + (b + a') * 1 =$$

$$a + (b + a') = a + (a' + b) = (a + a') + b = 1 + b = 1.$$

Using B2, B5, B4, domination, B1, B2, B4, and domination. Also,

$$(a + b)(a' b') = aa' b' + ba' b' = 0 * b' + a' bb' = 0 * b' + a' * 0 = 0 + 0 = 0$$

This time we used B5, B4, B3 and B1, B4, domination, and finally B3.

Hence, $a' b'$ is a complement of $(a + b)$; so is $(a + b)'$. So by Theorem 2, $(a + b)' = a' b'$.

Exercise: Use this same kind of argument to show that DeMorgan part (ii) holds.

It turns out that all of the theorems that we have shown using the Boolean algebra definition and even the axioms of a BA can be illustrated using truth tables for the **special case** of a BA that most interests us right now, the two-value $\{0, 1\}$ Boolean algebra of digital logic corresponding to {off, on}, or {FALSE, TRUE}, {LOW VOLTAGE, HIGH VOLTAGE}, {NO, YES}, etc., that we find in computer circuits. We demonstrated this fact for the unusual-looking distributive law (B5 #1) earlier. Some of our most interesting and useful problem-solving techniques involve truth tables, and we now return to their use with some examples and (in the next section) a powerful technique for simplifying Boolean functions. The theory of BA's and their generalizations is rich and beautiful and has given rise to some of the most celebrated theoretical and applied mathematics of the last century.

Exercise: Show the first statement of the DeMorgan Laws using a truth table rather than the BA axioms.

A systematic approach to logic functions

We look at some practical examples of the use of digital logic circuits.

Example 1: Seat belt warning light

Here is an example of a practical circuit, a seat belt warning light:

A: 0 if ignition is off, 1 if ignition is on

B: 0 if seat belt is unfastened, 1 if seat belt is fastened

Truth table

A	B	Seat belt buzzer	Terms in A and B
0	0	0	A'B'
0	1	0	A'B
1	0	1	AB' *
1	1	0	AB

Hence, we have that

F = A B' (A and not B --- corresponding to the row with the *)

The same truth table applies for the compare function of **A > B** (as in Column 5 of Figure 8-29).

Example 2: determining if a binary number is even.

Let $n = abc$ where a, b, c are bits (0's or 1's).

N in decimal	n in binary			n = abc is even
	a	b	c	
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

Figure 8-4 Truth table for F: "n is even"

If we let F be the statement, "n = abc is even," we can construct the function for the truth table in a very mechanical way. In this case, we should get a true outcome in 4 different ways from Figure 8-32. They are:

n = 0:	a, b, and c are false (have values of 0):	a'b'c'
n = 2:	a is false, b is true and c is false:	a'bc'
n = 4:	a is true, b is false, and c is false:	ab'c'
n = 6:	a and b are true and c is false:	abc'

If you can see a simple solution to this logic problem, just hold the thought as we go through the exercise.

Since n (going from 0 through 7) is even provided n = 0, OR n = 2, OR n = 4, OR n = 6, we can set up the function F by using the OR connective (+). Hence, F is true if $F = 1$. That is, we want:

$$F = a' b' c' + a' b c' + a b' c' + a b c = 1$$

This is a perfectly valid statement of the logic of the problem, but can it be simplified? Yes, using rules of Boolean algebra. In fact, this is the expression that we used in Example __ above. So, as we discovered, $F = c'$ is the simplified expression for this example. Does this make sense? Yes, since as we count from n = 0 to n = 7 the final bit (c) oscillates between 0 and 1, with the number n *even* precisely when $c = 0$, that is, when $c' = 1$.

The expression F given above is the sum of the terms given by the rows of the truth table having a, b, and c appearing in each of their regular and their primed forms. We call a row of the truth table a **minterm**. We can always find the expression for a function by adding the minterms corresponding to the rows for which the function F is true. This is exemplified in our choice of rows 0, 2, 4, 6 for the "n is even" function.

Example 3: does a three-bit binary number have an even number of bits?

n in binary			
a	b	c	Even no of 1's in n
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Exercise: Write an expression for the function F in the previous example, and simplify it using the axioms of Boolean algebra. [Recall that $xy' + x'y$ can be written as $x \oplus y$ and this is the complement of $xy + x'y'$, which is our function x iff y . An answer for this one can be $a \oplus (b \leftrightarrow c)$. There are likely other correct forms for the answer.]

Exercise: Find a truth table for which numbers from 0 to 15 are prime and write its function F as a sum of minterms.

Karnaugh Maps (Mapping the distributive laws to simplify Boolean functions)

It is often very convenient (and useful) to view the truth table for a Boolean function in a matrix form where the row- and column-headings contain combinations of the variables and their primes, and the cells contain the minterms corresponding to rows and columns.

In the following example, we illustrate the case where there are two variables and we use a matrix to designate the function

$$F = X' Y' + X Y'$$

Now the function F is shown in the left-hand table below in the form that we are used to. In the center table, we show another representation of this same information for F, namely, we fill in the cells in the appropriate rows and columns corresponding to the 1's in the left hand column for F, namely X'Y and XY'. The other two cells are blacked out. In the right hand table, we use 0's and 1's in those same places, where 0 corresponds to FALSE and 1 means TRUE.

X	Y	F
0	0	1
0	1	0
1	0	1
1	1	0

X \ Y	Y'	Y
X'	X'Y'	X'Y
X	X Y'	XY

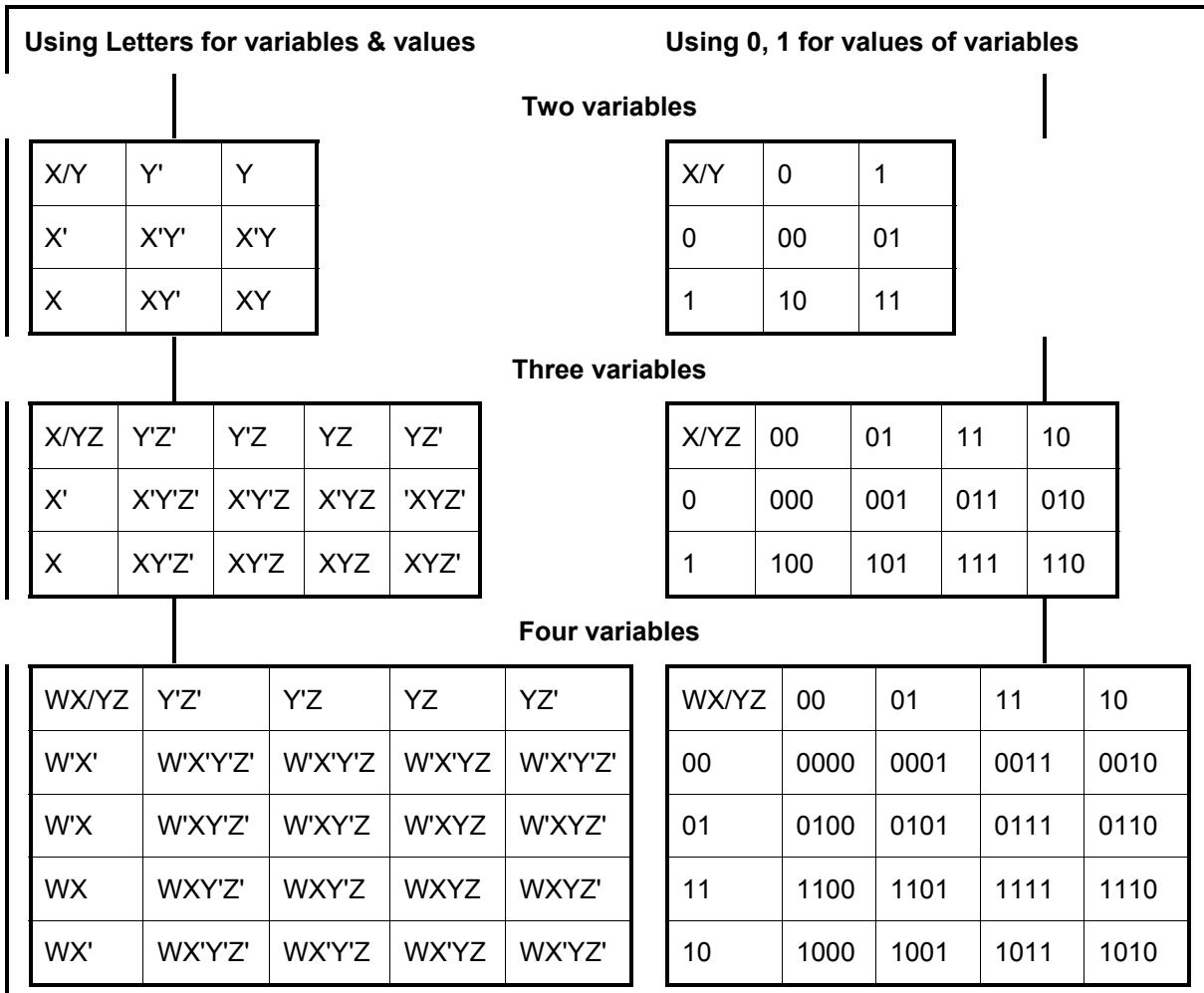
X \ Y	0	1
0	1	0
1	1	0

Now notice that the Y' column in the middle table (shaded) is filled in with minterms that completely pass through the X' and the X rows, so we can consider an "expanded" cell or a "rectangle" of two cells covering the cells containing X'Y' and XY'. This rectangle (which we see in the right-hand table having values of "1") completely fills in the Y' column. Notice, carefully, that the rectangle containing the 1's is precisely the column corresponding to Y' in the table on the right. Hence, we see that Y' is "covered" by 1's and so we can say that F is just

$$F = Y' \quad (\text{in Boolean algebra terms, } F = X' Y' + X Y' = (X' + X) Y' = 1 \cdot Y' = Y')$$

So the table (or Karnaugh map) did our distribution for us. In effect, the Karnaugh map is merely the distributive law carried out in a "visual" sense.

In the next figure, we show such matrix tables of this sort corresponding to 2-, 3-, and 4-variables, respectively. One can also work with 5 and even 6 variables, but these become more difficult to visualize and other, better techniques are available to deal with them.



The upper left cell in each map specifies the variables first along the left side and second as headers of the columns. So WX/YZ means that the headers of the ROWS will correspond to all the possibilities for listing W then X and their primes. Pay careful attention to the ordering of the values in the 3- and 4-variable maps. The columns and rows are labeled 00, 01, 11, 10 (namely, 0, 1, 3, 2) rather than in the usual numerical order. This order, called a gray code has the property that as you move from one entry to the next, the change from 0 to 1 or 1 to 0 happens at exactly one position, including going from the last entry to the first. This will prove to be crucial to us.

Now let's look at some specific examples of this important technique in action!

The situation: Is a 3-bit binary number $n = abc$ even or odd?

Let the functions F of the variables a , b , and c be equivalent to the statement: " $n = abc$ is an even binary number." We clearly saw its truth table in Example 2 above. We can now put that same data into a Karnaugh map in this way:

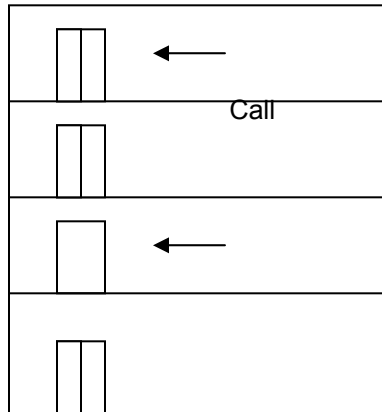
a/bc	00	01	11	10
0	1	0	0	1
1	1	0	0	1

Map for $F = \text{"abc is even"}$

In this case, we note that even though the shaded pairs of cells are separated (one on the left and one on the right), they have the property that going from the left column to the right column their column headings differ by just one bit. Hence, we consider them to be "adjacent" forming a 2×2 rectangle. In a sense, the table is wrapped around so that the left hand wall is "pasted" to the right hand wall making the shaded part truly a rectangle! Hence, since the only common thing in the two columns is a "0" in the c position and both a and a' are covered, this tells us that $F = c'$.

Case Study: the elevator controller

In this extended example, we want to build a logic device to control an elevator. The elevator is in a small building with only four floors that we will number in the European style as 0, 1 2, 3



Three logic devices will be constructed that will take as input the floor where the elevator is currently located and the floor where the call button was last pushed. There will be an up circuit, a down circuit and a stop (open door) circuit whose results depend upon how the call button floor and the elevator floor compare. Here is the initial data table. Look at it carefully to be sure you see exactly where the data come from.

Elevator at floor:	Call button pushed at floor:	UP	DOWN	STOP
3	3	0	0	1
3	2	0	1	0
3	1	0	1	0
3	0	0	1	0
2	3	1	0	0
2	2	0	0	1
2	1	0	1	0
2	0	0	1	0
1	3	1	0	0
1	2	1	0	0
1	1	0	0	1

1	0	0	1	0
0	3	1	0	0
0	2	1	0	0
0	1	1	0	0
0	0	0	0	1

If we rewrite the table with binary numbers instead:

Elevator at floor: ab	Call button pushed at floor: Cd	UP	DOWN	STOP
11	11	0	0	1
11	10	0	1	0
11	01	0	1	0
11	00	0	1	0
<u>10</u>	<u>11</u>	<u>1</u>	0	0
10	10	0	0	1
10	01	0	1	0
10	00	0	1	0
<u>01</u>	<u>11</u>	<u>1</u>	0	0
<u>01</u>	<u>10</u>	<u>1</u>	0	0
01	01	0	0	1
01	00	0	1	0
<u>00</u>	<u>11</u>	<u>1</u>	0	0
<u>00</u>	<u>10</u>	<u>1</u>	0	0
<u>00</u>	<u>01</u>	<u>1</u>	0	0
00	00	0	0	1

The formula for the elevator to go up, taken straight from the table, is:

$$UP = ab'cd + a'bcd + a'bcd' + a'b'cd + a'b'cd' + a'b'c'd$$

In diagram terms such as we introduced above, this looks like:

ab\cd	c'd'	c'd	cd	cd'
a'b'	0	a'b'c'd	a'b'cd	a'b'cd'
a'b	0	0	a'bcd	a'bcd'
ab	0	0	0	0
ab'	0	0	ab'cd	0

UP function shown as minterms

We could resort to some heavy duty algebra to find the minimal formula but the alternative, as discussed above, is the Karnaugh Map, which will make the simplification more transparent. To create a Karnaugh map, the table above is rearranged with **ab** values down the left side and **cd** values across the top-- 0's going in for FALSE values and 1's for TRUE values. Again, note the peculiar ordering of the binary numbers along each axis -- using a gray code. This type of ordering, using a gray code, turns out to be useful in a number of situations. Why it is useful here is that any two adjacent cells, adjacent either horizontally or vertically, differ by one variable going from **its value to its negative or the opposite!** In our example look at the two darkly shaded cells forming a rectangle designated by **a'b'c'd** and **a'b'cd**. These differ only in **c'** and **c**; which means that the formula could be reduced to **a'b'd** -- in Boolean calculation, this says that **a'b'c'd + a'b'cd = a'b'(c' + c)d = a'b' 1 d = a'b'd**.

The reference to **c** or **c'** drops out since we pass through a **c'**-column and a **c**-column in a rectangle of two adjacent cells. This is the basis of the Karnaugh map. We now convert these minterms in the cells by their **truth values** in the UP function.

The UP map

ab \ cd→	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

Truth values for the function UP

Here are the things that we notice in this particular map.

- Two 1x2 rectangles of ones (shaded in dark) implies that one variable can factor out by an application of the distributive law.
- A rectangle of 4 cells (4 cells in the upper right corner in this case) means that two variables will factor out by a double distribution.

The "adjacency" from top to bottom can wrap around the diagram.

In the UP map, three rectangles will suffice to cover all of the 1's in the chart. Look at each rectangle (or square) individually:

$ab \setminus cd \rightarrow$	00	01	11	10
00			1	1
01			1	1
11				
10				

The square $a'c$

The first rectangle (above) is a square where **d** changes from 1 to 0 and **b** changes from 0 to 1. That says that d and b will cancel out in the four terms covered by the square, by distributivity (try it!). If we look at the **a** and **c** values in these four cells, $a = 0$ and $c = 1$. The term that will cover all of these cells is thus

$ab \setminus cd \rightarrow$	00	01	11	10
00		1	1	
01				
11				
10				

The rectangle $a'b'd$

In the second rectangle (above), the one binary digit that changes is c going from 0 to 1. That says the term that will cover these two cells has no c term, it cancels out. The remaining binary digits are $a = 0$, $b = 0$, and $d = 1$, so the term is

$ab \setminus cd \rightarrow$	00	01	11	10
00			1	
01				
11				
10			1	

The rectangle $b'cd$

In the table above, we find one more rectangle that wraps around. In the wraparound, one binary digit changes and that is a going from 1 to 0. The other binary digits are $b = 0$, $c = 1$, and $d = 1$, so, the term that will cover these two cells is $b'cd$.

The complete, simplified formula is:

$$UP = a'c + a'b'd + b'cd$$

You will notice that this is considerably simpler than the "UP" formula that was given earlier. Simplicity here is important for several reasons. One is that it is easier to work with in doing calculations. Another is that in implementing the chip or circuit that controls when the elevator is to go up a floor, it takes a good many fewer electronic "gates" and so is cheaper to produce, works more rapidly, and generates less heat (so will probably last a longer time).

Exercise: find the minimized formulas for the Down and Stop buttons:

The DOWN map

$ab \setminus cd \rightarrow$	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

The simplified formula is ?:

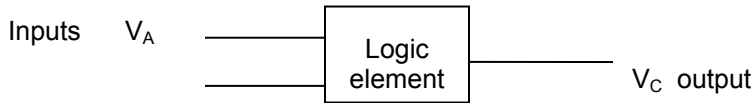
The STOP map

ab \ cd→	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

The simplified formula is ?:

Converting logic into a circuit

The logic operations described above can be converted into small circuit elements using transistors. In these circuits, an input bit of 1 is signaled by a voltage of 5 volts (5V) and a 0 bit is signaled by a voltage of 0 volts (0V).



A 2-input AND gate, to implement the AND truth table, would perform this operation

AND

X	Y	$X \wedge Y$
0V	0V	0V
0V	5V	0V
5V	0V	0V
5V	5V	5V

For our purposes, we will still use 0 and 1 for false and true, respectively.

The designer of a logic circuit can consider these objects to be “off-the-shelf” microchip units that can be purchased and assembled to build devices such as elevator controllers. The designer uses a program such as LogicWorks to drag and drop and connect an enormous variety of circuit elements to build and test a complete circuit.

The symbols depicted in Figure 8-33 are the common ones that denote the gates for the various logical connectives in designing electronic chips and components.

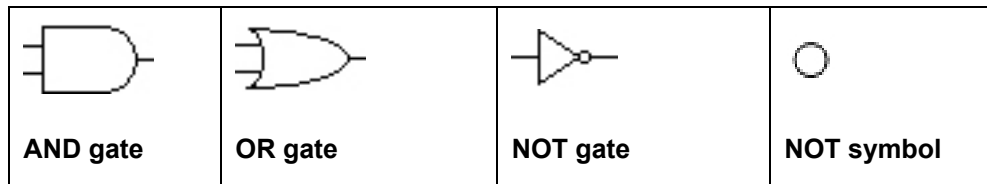


Figure 8-5 Common gates: AND, OR, NOT, and NOT symbol

The third gate is the NOT gate whereas the fourth symbol can be used in conjunction with other gates to indicate that a signal's value is to be reversed. We show an example of this in Figure 8-34 where signals a and b go into the configuration, b 's signal (passing through the small circle) is negated and the two resulting signals go into the OR gate giving us an output at $c = a + b'$. This kind of combination of gates and symbols can be put together to make up complex circuits.



Figure 8-6 The gate symbol for: $c = a + b$

Translating a logic formula to a circuit

Let's take the XOR formula derived earlier and convert it to a circuit

$$F = X * Y' + X' * Y$$

The * means AND while the + means OR. To build the circuit, we first take a couple of AND gates and connect them both to input lines holding the X and Y bits. Into one AND gate the Y bit should be NOTed and in the other the X bit should be NOTed. This can be done by a specific NOT unit or one can use an AND gate with a NOT unit incorporated. That is signified by a small circle at the input. The outputs from the two AND gates are connected to an OR gate to produce the final output.

Finally, to test circuit we need to be able to set either input to 1 or 0 by use of a binary switch, and we need to be able to measure the output bit with a binary probe that displays the bit.

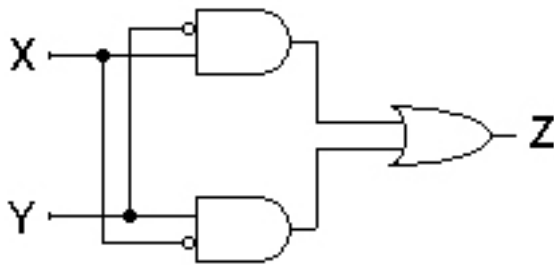


Figure 8-7 - logic circuit for XOR

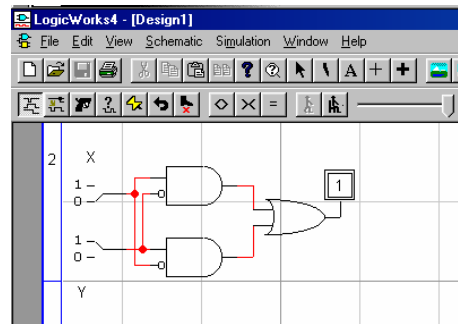


Figure 8-8 - logic circuit for XOR using the LogicWorks™ software

The UP formula for the elevator

The UP circuit derived for our elevator in the earlier case study.

$$a'c + a'b'd + b'cd$$

can be built using three AND gates. A couple of the AND gates show three inputs and the final connection to the OR gate also has three inputs. A multiway AND gate can be built by combining two inputs to get one output that is then combined with another input in a cascade. The combined gates will only produce a 1 if every input bit is 1. Rather than force us to build and

rebuild these cascades, multi-input AND gates are used. Similarly, multi-input OR gates produce a 1 for output if any input has a 1 value.

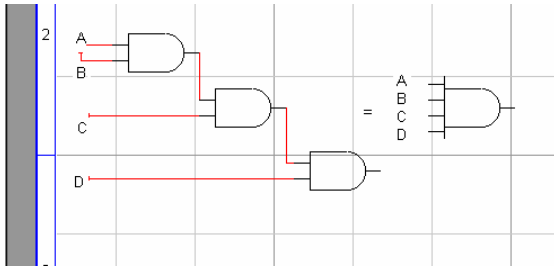


Figure 8-9 - multiway AND gate

In the circuit diagram for the UP circuit, you can see the three-input AND gates. Some of the inputs are to be negated and in some case we have explicitly used the NOT gate.

The circuit also use hex keypads to provide the ab and cd inputs. A hex keypad responds to a press of a hexadecimal digit , for example A, by outputting the 4 bits, 1010, from top to bottom. A 1 comes from the top line, a 0 from the next, a 1 from the next and a 0 from the bottom line.

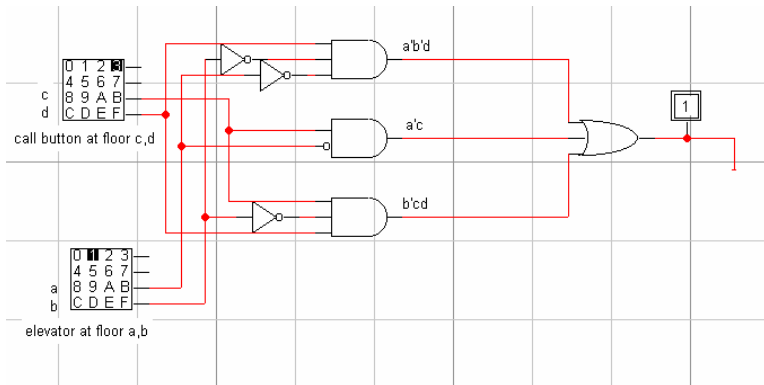
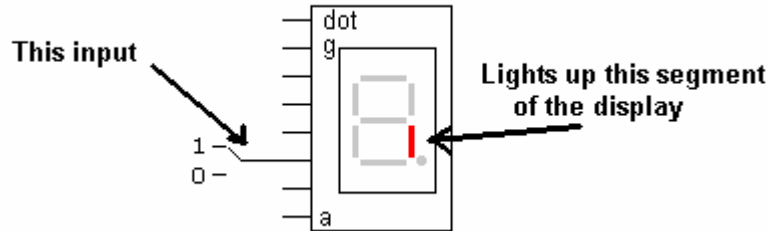


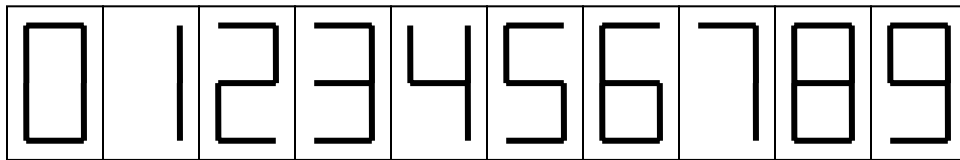
Figure 8-10 - UP circuit

Case Study: the 7-segment display

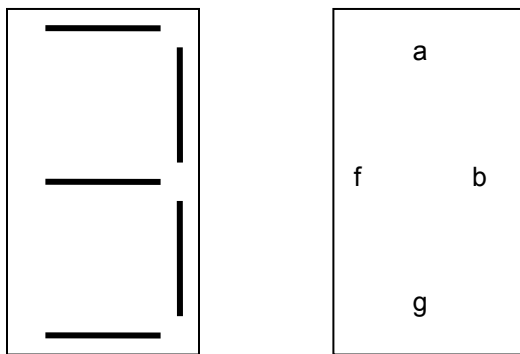
The 7-segment display is ubiquitous. It is the little device in that displays the numbers you encounter all the time on clocks, microwave ovens, televisions, everywhere.



The name comes from the number of segments commonly required to construct the number displays.



The number 3, for example, is displayed by lighting segments a, b, c, d, g and not lighting segments e, f.



Writing a control circuit for a lamp requires a slightly unexpected point of view. One doesn't write a circuit to produce 3 for example. Instead, one writes control circuits for each of the segments a-g so that, as the binary code for 3 comes into the circuit, each segment will determine whether or not it should light up. A truth table for all of the segments must first be constructed. Since at least 4 bits are necessary to contain numbers 0 through 9, a full set of 4-bit numbers 0000 to 1111 is used with 0 entered into the table below the 9 digit. If one wanted to build a circuit to display

hexadecimal digits, all 16 rows would have to be filled in. The input bits are labeled **A B C D**, not to be confused with the segments **a b c d e f g**.

decimal digit	binary ABCD	a	b	c	d	e	f	g
0	0000	1	1	1	1	1	1	0
1	0001	0	1	1	0	0	0	0
2	0010	1	1	0	1	1	0	1
3	0011	1	1	1	1	0	0	1
4	0100	0	1	1	0	0	1	1
5	0101	1	0	1	1	0	1	1
6	0110	1	0	1	1	1	1	1
7	0111	1	1	1	0	0	0	0
8	1000	1	1	1	1	1	1	1
9	1001	1	1	1	1	0	1	1
	1010	0	0	0	0	0	0	0
	1011	0	0	0	0	0	0	0
	1100	0	0	0	0	0	0	0
	1101	0	0	0	0	0	0	0
	1110	0	0	0	0	0	0	0
	1111	0	0	0	0	0	0	0

segment a

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	0	0	0	0
10	1	1	0	0

$$A'C+AB'C'+A'B'D'+A'BD$$

segment b

AB \ CD	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	0	0	0	0
10	1	1	0	0

$$A'B'+B'C'+A'C'D'+A'CD$$

segment c

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	0	0	0	0
10	1	1	0	0

$$A'B+B'C'+A'D$$

segment d

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	0	0	0	0
10	1	1	0	0

$$B'C'D'+A'B'C+A'CD'$$

$$+A'B'C'+AB'C'+A'BC'D$$

segment e

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	0	0	0	0
10	1	0	0	0

$$B'C'D' + A'CD'$$

segment f

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	0	0	0	0
10	1	1	0	0

$$A'C'D' + AB'C' + A'BC' + A'BD'$$

segment g

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	0	0	0	0
10	1	1	0	0

$$A'BC' + AB'C' + A'B'C + A'CD'$$

To build a circuit and test it, we will use LogicWorks

Here is one further detail for consideration. The circuit as set up will not respond to input values of 10 to 15 (A to F in hexadecimal). If we could be certain that such values would never appear as input, easy enough to guarantee in a program, then it would not be necessary to set the truth table to 0 for the values A to F. Strictly speaking, we don't care. So, in that case, we could use a symbol *d* to indicate that it doesn't matter if the value is 0 or 1. Why do that? Because in some cases by using a 1, the logic formula might be simplified.

segment a

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	d	d	d	d
10	1	1	d	d

$$C+AC'+A'B'+BD$$

segment b

AB \ CD	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	d	d	d	d
10	1	1	d	d

$$A'B'+B'C'+C'D'+A'CD$$

segment c

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	d	d	d	d
10	1	1	d	d

$$A'B+B'C'+A'D$$

segment d

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	d	d	d	d
10	1	1	d	d

$$B'C'D'+A'B'C+CD'+A'B'C'+AB'C'+A'BC'D$$

segment e

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	d	d	d	d
10	1	1	d	d

$$B'C'D' + A'CD'$$

segment f

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	d	d	d	d
10	1	1	d	d

$$A'C'D' + AB'C' + A'BC' + A'BD'$$

segment g

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	d	d	d	d
10	1	1	d	d

$$A'BC' + AB'C' + A'B'C + A'CD'$$

Turning logic circuits into a computer

Now that we have used digital logic to implement some control systems, let's carry it one large step further and look at the process of building the most general purpose control system, a computer. In this section, we will develop a few component circuits that are key to a computer's functioning, link a few of them together, and then ask you to imagine how the whole process could be scaled up to a 30-million transistor CPU chip.

In the diagrams below, we'll use these symbols for the common gates:

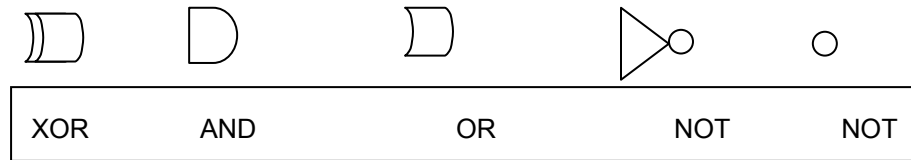


Figure 8-11 - symbols for gates used in circuit diagrams

An adder

The simplest question perhaps is how one does arithmetic with logic gates. In particular, how can one add two numbers? Consider the truth table with the results of adding two bits. Two results occur when adding two bits – the sum and a possible bit to carry into the next column.

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		XOR	AND

The circuit for the combined truth tables is

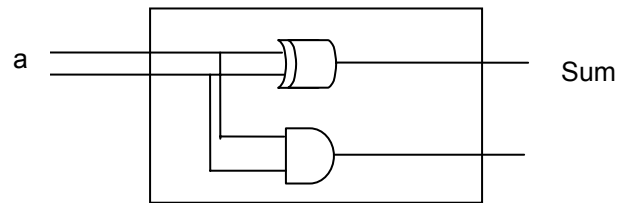


Figure 8-12 - half adder

The circuit in Figure 8-40 - half adder is called a half adder because it does not take into account a possible incoming carry bit from a previous column. The next circuit takes care of that omission.

Full adder

A full adder, on the other hand, actually adds 3 bits – the two bits we have plus a bit being carried:

A bit	B bit	Carry in bit	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

and is implemented by using two half adders

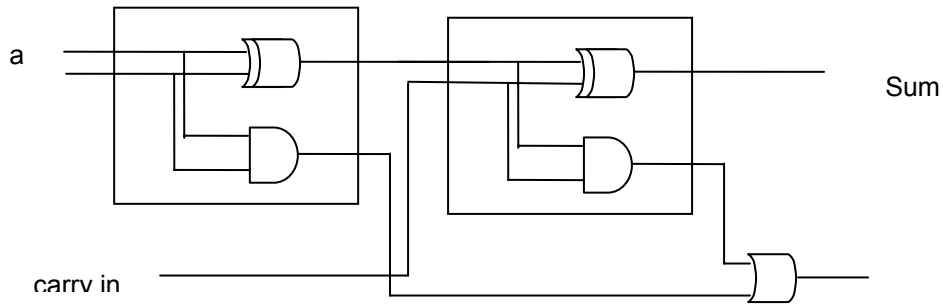


Figure 8-13 - Full Adder

A ripple adder

To add two 32-bit numbers requires nothing more than stringing 32 full adders together with the carry bit from each adder used as an input to the next. Actually, it is a bit more difficult than that. Computer circuits work with timing cycles. The 800 MHz or 3.2 GHz that your CPU is rated at is the clock speed, the measure of how of fast instructions are processed. If 32 adder units were simply strung together, the time it took for the logic to propagate from one end of the adder pipeline to the other would exceed the time allowed between clock cycles. Logic designers have devised shortcuts to work around these kinds of problems but that's for another course. For now, just assume we could build a computer with a ripple adder.

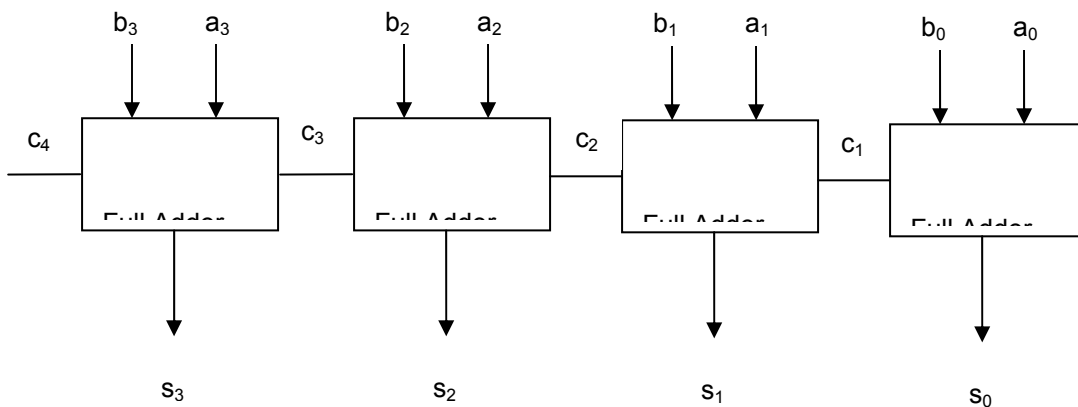


Figure 8-14 - a 4-bit ripple adder

Memory

The D latch in Figure 8-43 is an interesting device that introduces the concept of a sequential circuit.

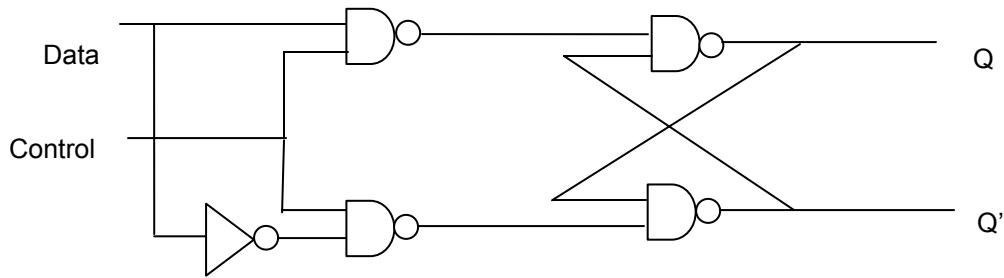


Figure 8-15 - The D Latch

First, experiment with the D Latch. Suppose that currently the two outputs of the system Q and Q' are either 0 or 1 but we don't know which. Now, let C (Control) be 1 and D (Data) be 0. What values do Q and Q' end up with?

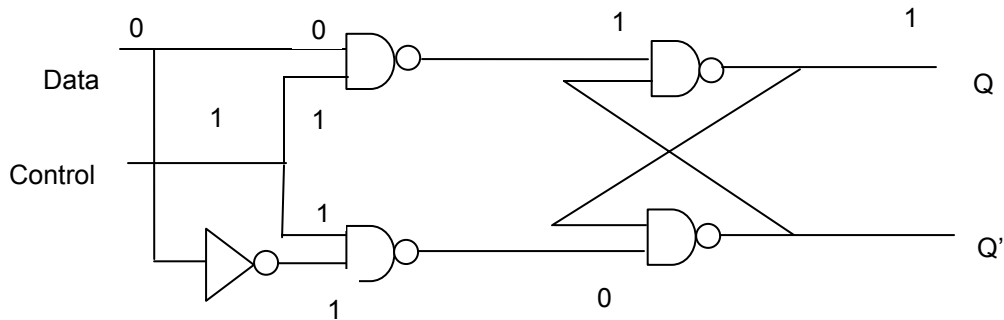


Figure 8-16 - The D Latch with D= 0, C=1

If Q = 1 and Q' = 1 to start, the input to the lower right NAND gate is 1 and the output then become 1; the input to the upper right NAND gate is 1 and the output becomes 0. If the 0 output for Q now appears at the lower left NAND, the output remains 1 and the upper gate remains at 0.

If Q = 0 and Q' = 0 to start, the input to the lower right NAND gate is 0 and the output then become 1; the input to the upper right NAND gate is 1 and the output becomes 0. If the 0 output for Q now appears at the lower left NAND, the output remains 1 and the upper gate remains at 0.

Suppose now the input D is switched to 1.

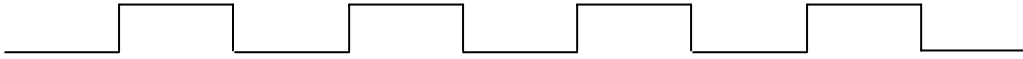
Now, switch the C input to 0 and notice what happens to Q and Q'.

Finally, switch D again to 0 and notice the Q and Q' outputs.

The point of all of this is that the latch retains a memory of the value of D (as the output Q) even if D changes as long as C is off; On the other hand, when C is on, Q will change when D changes. In other words, this D latch can serve as a 1-bit memory element! The memory element can have its value changed only when the control bit is 1.

An actual memory element would have two controls. One control would be part of the addressing scheme. A bit value of 1 would say this is the memory element to set. The second control would be the clock. That control can work in either of two ways. In one form, the data can be set only when the clock signal is set high (value = 1); in the other form, the signal can only change when the clock is going from high to low (value changing from 1 to 0).

So, effectively, a clock marches data through the system



Whenever the clock is high, data values are reset and new values are then ready for the next round of input to the logic elements like the arithmetic unit.

Multiplexer

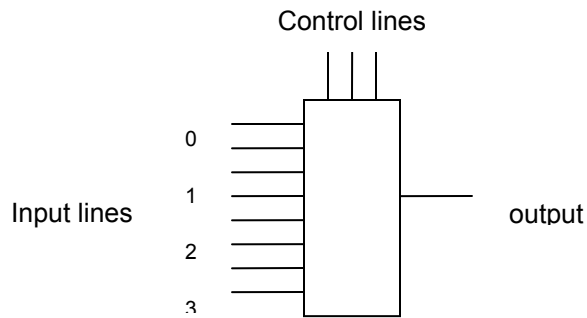


Figure 8-17 8-to-1 multiplexer as black box

The output for the multiplexer in Figure 8-45 is just the input from the one of the 8 input lines selected via 3-bit number on the control lines. Think of the input from the control lines as representing a binary number, let's say 1 0 1 or 5. The output will then be the input from line 5, whether it is 0 or 1.

To understand the workings of the multiplexer, consider two cases in the simpler 4-to-1 version in Figure 8-46. Assume the control lines have the binary value 11 (or 3) and that there is a 1 bit on input line 2. What value appears on the input line? Then assume a 1 bit on input line 3. What value appears on the output line?

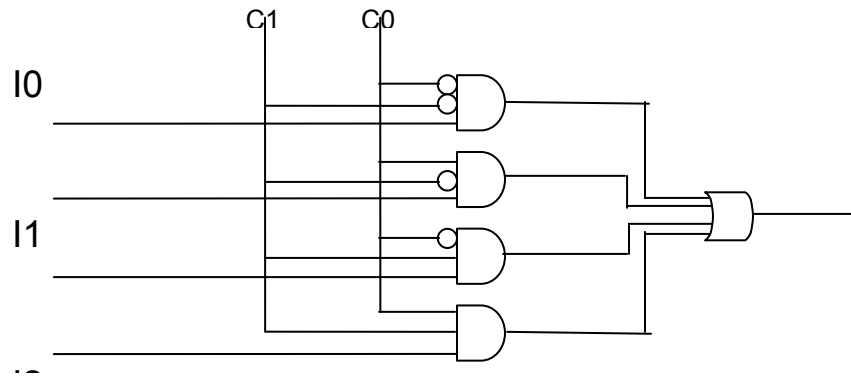


Figure 8-18 - 4-to-1 multiplexer built from gates

Decoder

A decoder is sort of the opposite: send it a 3- bit number and it will send a 1 out the corresponding output line

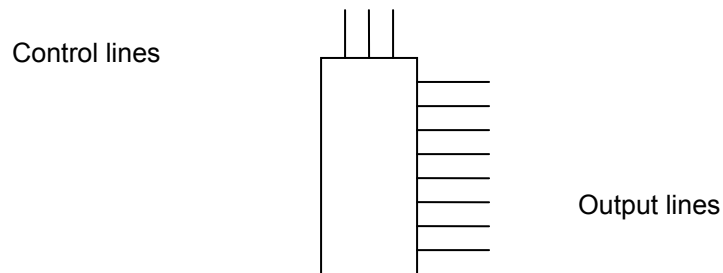


Figure 8-19 - 3-bit decoder as black box

Again for the sake of simplicity, consider the 2-bit decoder below and work out the output on each line for the 4 possible control line inputs in the table below

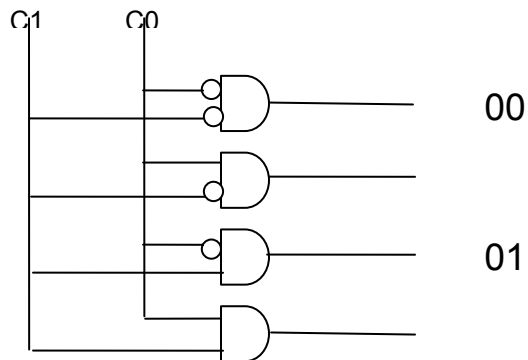


Figure 8-20 - 2-bit decoder built from logic gates

C1	C0	O0	O1	O2	O3
0	0				
0	1				
1	0				
1	1				

Setting a 1-bit register

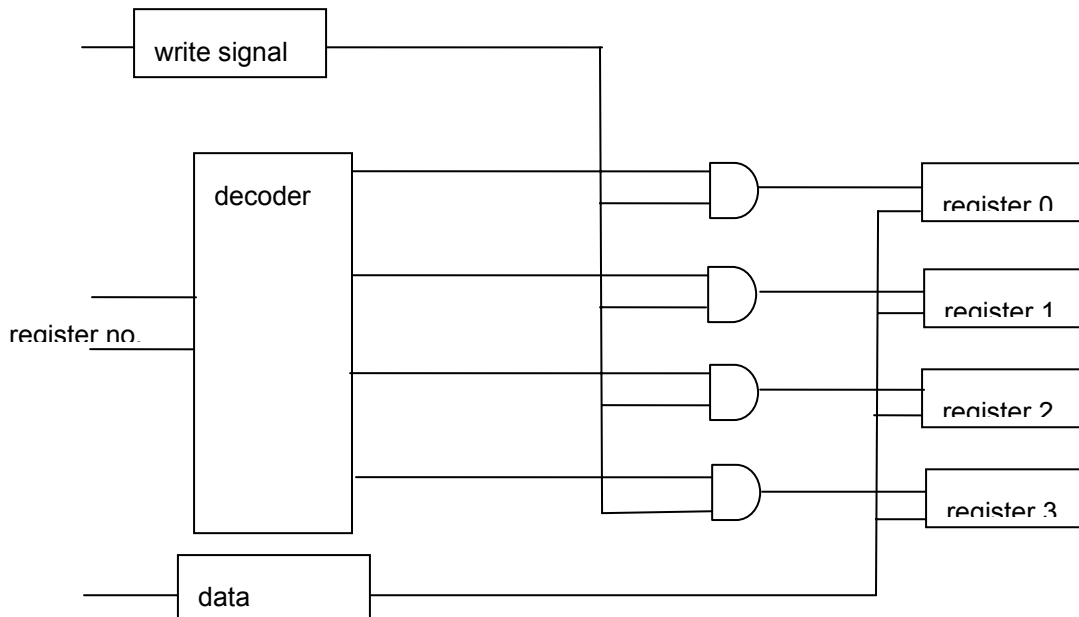


Figure 8-21 - The circuit to set a 1-bit register

Now, with all the pieces, consider how the circuit in Figure 8-49 would work. Its purpose is to take one bit of data coming in on the data line and store that piece of data (0 or 1) in the particular register that we have chosen. The register value is coming in on the two lines labeled **register no.** and can refer to registers 0, 1, 2, or 3 (00, 01, 10, 11).

The decoder takes the register number input and puts a 1 on whichever line has been chosen and 0 on every other line. Now, the data will present itself at every register, but the *only registers that can change will be those for which a write signal is also present*. The only register to have a write signal will be the one for which a 1 has been ANDed with the write signal, so, only the register we have chosen gets changed.

How a one bit arithmetic logic unit works

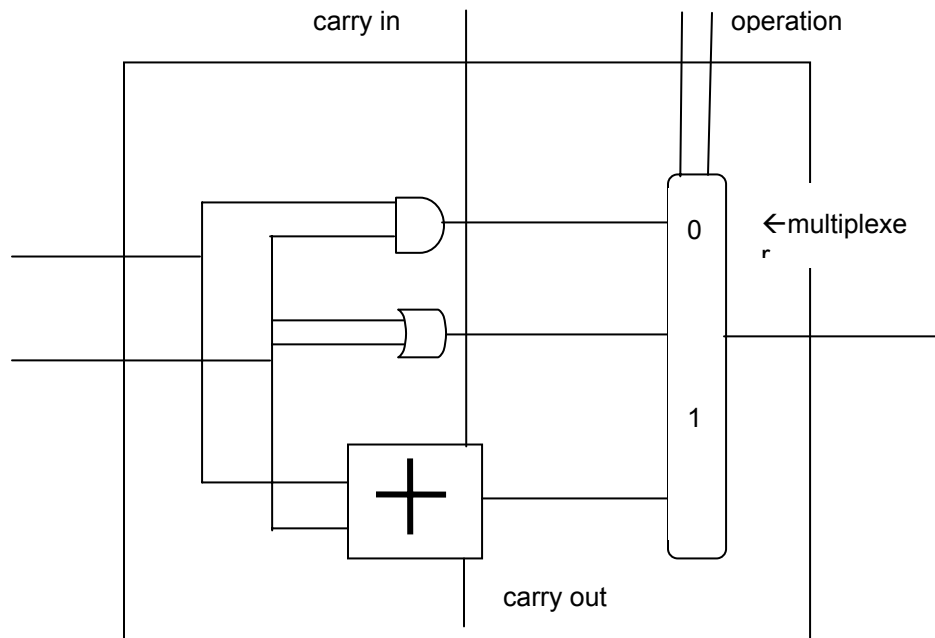


Figure 8-22 - A 1-bit arithmetic unit

We can do a similar analysis to look at the workings of a small **arithmetic logic unit (ALU)**, the heart of a CPU. This arithmetic logic unit is pretty minimal. It can only operate on one bit and can do exactly three possible operations: add two bits, do an OR operation on two bits, or do an AND operation on two bits. Notice the interesting fact that all of the operations are actually carried out but the multiplexer **CHOOSES** which answer is passed on according to the input sent into the multiplexer over the lines labeled operation.

How does this relate to the real chips you find in real computers? The pieces are all here, but require some imagination. First imagine expanding the ALU to include many, many more operations. That would require a multiplexer with many more lines, but that's not difficult to imagine. Then, use the same kind of idea discussed with the ripple adder to imagine taking say 32 of these ALUs and ganging them together to build a unit capable of handling 32-bit data.

Now, an actual computer instruction would consist of say a 16-bit word that would contain an instruction that, in English, might mean add (the contents of two special registers) and put the result in memory location 376. The instruction itself, translated into machine language, would be simply a string of bits. The first 4 bits, for example, could represent the operation to be performed (ADD) and those bits would be fed into the ALU above to choose the operation. The last 12 bits could contain the address of the memory where the answer should go. Those bits would be routed to the circuits that set register values (Figure 8-49 - The circuit to set a 1-bit register). So, on a given clock cycle the data contained in two special registers would be added and presented to all of the registers but only stored in the one with a write signal also present.

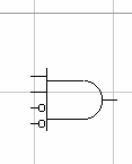
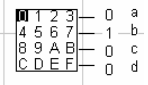
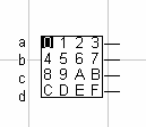
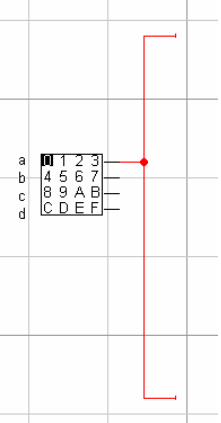
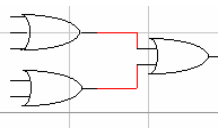
Now, of course, if you go back and look at the schematic of a computer from chapter 1, there are some additional pieces including registers, cache memory, regular memory, buffer for keyboards, graphics boards, and on and on, but once again it is a matter of starting with some simple pieces and just putting more and more of those pieces together into more and more complex structures.

Some tips for working with LogicWorks

For purpose of this example, imagine that you have developed a function like:

$$a'bc + a'b'cd + ab + bcd'$$

To retain your sanity, do these steps:

<p>Create a label for each of the terms and spread them out vertically down the middle of the LogicWorks design sheet, where all of the AND gates will reside. Leave enough space between the labels</p>	<p>a'bc a'b'cd ab bcd</p>
<p>Put the appropriate AND gate near each label. How do you determine the appropriate AND gate? Choose an AND gate that has as many inputs as the number of letters in the term and as many INVERTed inputs (NOT gates) as there are 'primed' letters in the term. For example, the term a'b'cd would require a 4-branched AND with 2-inv</p>	
<p>Put the hex keypad to the left side of the design sheet. Remember that if you have a number like 0100, with the digits called a b c d, the outputs from the keypad will be →</p>	
<p>So, put a label on the left of the keypad with a b c d in a vertical line.</p>	
<p>Now, start with the “a” line. Draw a short line horizontally out from the keypad and then join to it a vertical line running from the top AND gate to the bottom. Draw horizontal line from the AND inputs to the vertical line, using a NOT input line wherever a' appears in a formula and a regular input line where a appears in the formula</p> <p>Repeat for the other lines, offsetting them to the right.</p>	
<p>Join the AND gate outputs to an OR gate or more than one OR gate which you then combine with another OR, if you can't find an OR gate with the correct number of inputs</p>	

Sidebar II: Boolean Algebra and Cryptography:

None of your business!

Applying XOR

A major concern of governments, corporations and individuals is to keep private information private. This can be especially difficult if the information is on a computer that is accessible via a wide area network or if the information needs to be transmitted. The answer to the security dilemma is to encrypt the information, to convert it into a form that can only be read by someone who has the key to deciphering the information.

The basis of cryptography is the simple XOR operation. Imagine that we have a message, or a document, or any piece of information that can be stored as a stream of bits. In this case, consider the simple message:

“Hide this message!”

The plaintext segment below is simply the message written as ASCII code. Try matching the code to the letters:

plaintext: Hide this message!

```
01001000011010010110010001100101001000000111010001101000011010010111001100100
0000110110101100101011100110111001101100001011001110110010100100001
```

Next we need a key that can be used both to encipher and decipher the message. We will use the simple expedient of using a keyword and once again will convert the key to ASCII code.

key word: secret

```
011100110110010101100011011100100110010101110100
```

The encryption is a very simple process. The key is lined up with the plaintext and a bit-by-bit (bitwise) XOR operation is carried out between the plaintext bits and the bits in the key. When we run out of key, we just start over at the beginning of the key once more. Check that the cipher presented below is indeed the result of XOR of the plaintext and the key for the first ten bits.

ciphertext

```
00111011000011000000000100000000010001010001000100001101000011000001011001000
1010000100000000000000010110000101100000010000000010000000001000100
```

Now, here is the marvelous thing about XOR. If you apply the XOR operation with the same key to the cipher, you get the plain text back again.

plaintext

```
01001000011010010110010001100101001000000111010001101000011010010111001100100
0000110110101100101011100110111001101100001011001110110010100100001
```

Hide this message!

Strong encryption

That seems an easy enough process for encryption. The problem is that it is just as easy for an experienced cryptographer to decipher the message, especially if there is a good quantity of encrypted text to work with. The cryptographer can look for repeating patterns, eventually figure out the length of the key, and apply myriad tools to breaking the code. The problem here is that there is a clear link between any encrypted bit and the key bit that was used to encrypt it and eventually the cryptographer can find the connection.

Serious cryptographic algorithms work to hide the connection between a bit and the bit that encoded it. For example, the DES algorithm (Data Encryption Standard) that was the basis for commercial encryption until very recently does a very complex dance with the bits. A 56-bit key was used to encrypt a 64-bit chunk of data at a time. The 64-bit plaintext was divided into two halves, each half expanded, then the key (which will have been shifted) is applied, a complex substitution process is applied to the bits, the results are permuted, the left and right halves are swapped and the process is repeated – a total of 16 times. The result is that at the end there is virtually no dependence of a particular enciphered bit on a particular key bit.

Even this complex algorithm had become vulnerable to breaking with the development of higher and higher speed computers so a few years ago the National Institute of Science and Technology announced a competition to replace the DES standard with a new Advanced Encryption Standard (AES). In 2000 the final winner of the competition was announced, the Rijndael algorithm developed by a pair of Belgian (Flemish) cryptography specialists. This was a bit of a surprise, apparently, or as the creators, Dr. Joan Daemen and Dr. Vincent Rijmen, say “It’s a classic story of two guys in a garage taking on the establishment and winning.” If you want to find out more about the complexity of the new encryption standard, go the Rijndael Page

<http://www.esat.kuleuven.ac.be/~rijmen/rijndael/>

and download the tutorial and animation zip files. Plus, the page is fun to read.

Strong encryption algorithms turn out to take a lot of time. If an encryption program is applied to a stream of bits being sent out to a communications network, the software won’t be able to keep up with the transmission rate. For this reason, the algorithms are usually converted into hardware, special-purpose chips designed just to encode and decode.

To be even more secure, the keys need to be changed often so that someone intercepting the messages does not gather enough information enciphered with one key. But, that points up another issue of security – distributing the keys.

Public Key Algorithms

An ingenious concept called a public key algorithm gets around this problem. It can be used to encrypt documents but it is a slow algorithm and would not be used for high-speed data encryption. More likely one would use a public key algorithm for low-speed information transfer (mailing a file, for example, that you’d like only the recipient to read) and, more importantly, it can be used to deliver the keys to the faster algorithms.

The idea is this. I will publish, i.e. let everyone know, my public encryption key. The key will have the remarkable feature that if **you** use **my** key to encrypt a message, I am the only one who can decrypt it. **My** key that **you** used to encrypt the message will not decrypt it. My secret key will.

So, there are two keys, the public key that anyone can use to send me a message and the private key I use to decrypt it. Naturally there must be some connection between the keys. There is and it is subtle. Here is how the keys are created:

- I choose 2 very large prime numbers p and q (these are both kept secret) and use those numbers to compute $N = p \cdot q$
- I then choose the public encryption key e such that e and $(p-1) \cdot (q-1)$ are relatively prime.
- I publish e and N (wait, if someone knows N , can't they just factor it into p and q ? No, this turns out to be an extremely difficult problem. See below)
- I also find d such that $e \cdot d = 1 \pmod{(p-1) \cdot (q-1)}$. This looks very difficult but there are simple algorithms to do this. Notice that d can only be found if p and q are known, and only I know them.

To encrypt a message that you want to send to me, you break it into small numbers M_i and compute the encrypted values of the numbers:

$$C_i = (M_i)^e \pmod n$$

Then it turns out that I can decrypt the number back into the original number with this calculation:

$$M_i = (C_i)^d \pmod n$$

Nobody except me knows the value of d . You might think that someone could factor N to find p and q , and so find d given e . But, that is not the case. Breaking a number into its constituent primes is a very complex task that increases in difficulty rapidly as the size of the number increases. It isn't possible to try all combinations of primes, either. There are approximately 10^{151} primes of 512 bits or less.

The key to breaking public key cryptography is to be able to factor very large numbers, a task that presently looks impossible *with computers as they are now*.

A last interesting point about cryptography is that governments and individuals don't have the same goals with respect to cryptography. Governments would like to see internal communications kept private but don't necessarily want that power in the hands of individuals, especially those engaged in criminal or threatening acts. For that reason, governments typically have discouraged or restricted the export of cryptographic algorithms, putting them in the same category as munitions. Individuals, even the law-abiding ones, on the other hand, are often not eager to have everything they communicate available to the government. Some have acted on their principles and developed algorithms and tools and made them freely and publicly available to provide them to anyone who wants cryptographic tools. The best known example is Pretty Good Privacy (PGP).