F E A T U R E

Designing with Logic, Part 1

Even the latest generation of microprocessors with zillions of transistors on a chip are, at heart, based on simple binary logic. This month we'll start looking at some of the principals behind this essential area of electronics.

Steve Rimmer

Digital logic is often among the most misunderstood areas of electronic applications. If you come from a background of analog circuitry, logic design can be baffling. While logic circuitry is based on the same sorts of devices as other types of circuits are, the input and output of a logic circuit consists of connections of states rather than of signals *per se*. Conventional approaches to design don't really work when they're applied to logic.

It's often possible to design and debug logic circuits without ever powering up an oscilloscope.

For the next few months, we're going to look at the basics of computer logic. Logic design can be applied to simple circuits which just happen to use logical elements as well as to complex hardware projects specifically intended for use with microcomputers.

The Gate

There are relatively few essential logical devices, and, as we'll see in the coming months, many of the seemingly complex logical elements which hardware designers use as integrated devices are really just arrays of simple logical elements inside. Part of the usefulness of logic is its predisposition for creating increasingly more complex and functional "black boxes".

Logic deals with "binary states". To keep the discussion simple, and in familiar electrical terms, we'll allow that a binary state is a voltage level. The level zero is represented by zero volts. The level one is represented by five volts. Different logic families treat these values differently, but we'll be talking about generic logical concepts here.

In academic circles, a logic state of zero is referred to as being "false". A logic state of one is referred to as being "true". This will crop up later on.

A single logic state doesn't tell you very much, inasmuch as it can only be in one of two states. The usefulness of logic is in having multiple elements, each with its own independent state.

The simplest logical element is a NOT gate, or "inverter". This is a box which complements the state of its input. If you apply a state of one to its input, its output will be zero. Its logical symbol is illustrated in figure one.

In fact, this device can be seen as the combination of two still simpler elements. The triangular bit is a buffer and the dot at the output is the thing that complements the output of the buffer. In logical terms there is never any need for a buffer, but in practical electronic applications logical signals frequently need to be buffered.

Although it's a bit simplistic at this state, we can represent the functioning of the NOT gate with a truth table. This rational can be applied to all logical elements, and it will turn up as a design tool when we go to actually connecting the logical elements together. Here's the truth table for a NOT gate, or inverter.

INPUTOUTPUT
10
01

The next simplest logical element... or, at least, the next most commonly used one... is the AND gate. This is an element which accepts two inputs and produces one output. It works on the rule that if both of the inputs of the gate are high, its output will also be high. Otherwise, it will be low. In logical terms, we would say that if input one AND input two are true, the result of the process... the gate... will be true.

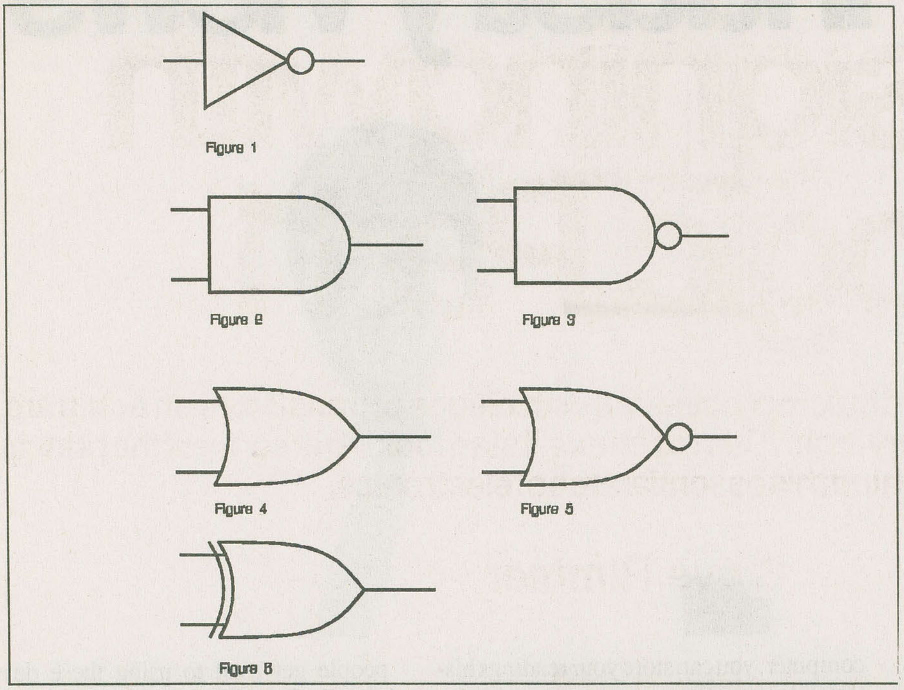
Figure two illustrates the symbol for an AND gate.

This is the truth table for an AND gate.

INPUT 1INPUT 2OUTPUT
000
100
010
111

We can create another gate from this one very simply by adding a dot to the output. The dot, as you will recall from the discussion of the inverter, NOTs everything that passes through it. We call the resulting gate a NAND gate. Its truth table is the inverse of that of the AND gate.

INPUT 1INPUT 2OUTPUT
001
101
011
110



If you've been following the C language series which has been running in this magazine for the past few months, you'll recognize the foregoing truth tables. They operate the same way as does the bitwise arithmetic under C.

The next sort of gate we'll encounter is the OR gate. Its logical symbol is illustrated in figure three. It works under the rule that if either of its two inputs are true, its output will be true. We can write its truth table like this.

INPUT 1INPUT2OUTPUT
000
101
011
111

Like the NAND gate, the OR gate will also spawn a negative clone of itself if we tack a dot onto its nose. The NOR gate is shown in figure four. Its truth table, predictably, is the complement of the OR gate truth table.

INPUT 1INPUT2OUTPUT
001
100
010
110

NAND gates turn out to be very useful logical elements in designing complex logic circuits. NOR gates are much less frequently encountered.

The final sort of gate to be discussed is the exclusive OR gate, or XOR gate. It has a true output if one but not both of its inputs is true. Its logical symbol is illustrated in figure five. Its truth table goes like this.

INPUT 1INPUT2OUTPUT
000
101
011
110

Wiring

Designing with logical elements simply involves putting these gates together to make up a logic array that does what you want. This is a bit like saying that playing classical violin is simply a matter of putting your fingers on the right strings at the right time and moving the bow.

While we will discuss binary arithmetic in much greater detail later in this series, let's have a look at the basis of it now. Dealing with numbers in this way will help you to understand how logical elements are employed to work with numeric values.

Binary numbers are represented as collections of true and false states. For this example, we'll deal with binary numbers from zero through three. These are said to be "two bit" numbers, because they can be represented by two states. We'll call these two state elements line zero and line one. This is how they represent the first four numbers.

LINE 0LINE 1VALUE
000
101
012
113

The value of line zero is said to be one. The value of line one is two. These are, more properly, one raised to the power of zero in the first case and one raised to the power of one in the second.

Let's create a hypothetical logic element called ADD. This is a fictional gate with two inputs and two outputs. It will add binary values, although as yet we do not know how it works. It has four input lines and four outputs, that is, it will accept two two-bit binary numbers as input and spit out a two bit result. In fact, we need an additional output line for a carry, to be used as a flag should the output exceed the values which are legal for a two bit number.

The truth table for this element would be as follows.

INPUT 1INPUT2OUTPUT
LINE 0LINE 1LINE 0LINE 1LINE 0LINE 1CARRY
0000000
1000100
0100010
1100110
1110001
1101101
1111011
0000000
0010100
0001010
0011110
1011001
0111101
1111011

You might want to see if this thing's binary values actually work out right, that is, if for example the result of this binary calculation

LINE 0LINE 1

01
plus10
equals11

is actually correct in decimal terms. Let's see how that works. The first input has the decimal value of two. The second input has the decimal value of one. One plus two is three, or at least is was when I checked last. The decimal result of having both lines of a two bit number true is, in fact, three.

Next month we'll design the actual logic array for the ADD element, as well as looking at some additional binary math. ■