Tutor

BY JEFF PROSISE

It All Adds Up: How Computers Do Arithmetic

We no longer think of computers as machines that simply *compute*, or calculate. But the simple arithmetic tasks—addition, subtraction, multiplication, and division—are essential to any computer's operation. This is true whether you're calculating a spreadsheet, displaying a new font in your word processor, or playing a sound-and-video clip.

If you read this column in our issue of January 10, 1995, you learned how dynamic random access memory (DRAM) works. One DRAM cell, which stores a single bit of information in computer memory, consists of a transistor and a capacitor. The capacitor stores the bit's value; a charge in the capacitor represents a 1, while a lack of charge represents 0. The transistor acts as a miniature switch. When closed, the switch completes a circuit between the capacitor and a conductor known as a bit line, allowing the capacitor's charge to be read or written. Put millions of these tiny cells together and throw in some supporting circuitry, and you have the equivalent of a modern-day DRAM chip or SIMM.

This time around we're going to investigate how a computer does arithmetic—specifically, how it adds two numbers and comes up with a result. We'll start by reviewing the binary numbering system and performing some simple binary addition by hand. Then we'll look at the specialized circuitry that the microprocessor inside your PC uses to add 1s and 0s. Finally, we'll see how simple circuit blocks are put together to form full-blown adders. Best of all, you won't need an electrical engineering degree to follow along. If you can understand the difference between on and off, you can understand how computers do arithmetic. No kidding!

THINKING IN 1s AND 0s Most everyone knows that digital computers think in 1s and 0s. Using the base-2, or *binary*, numbering system, a computer represents large numbers with strings of 1s and 0s. Here's how a computer would count from 1 to 10 in binary:

```
    1
   10
   11
  100
  101
  110
  111
 1000
 1001
 1010
```

Each "place" in a binary number represents a power of 2, just as each place in a decimal number represents a power of 10. The digits in a 3-digit decimal number, for example, specify, from right to left, ones, tens, and hundreds. Similarly, the digits in a 3-digit binary number signify ones, twos, and fours. Hence, the binary number 101 equals (4*1) + (2*0) + (1*1), which equals 5.

You can add binary numbers just as you add decimal numbers: Line them up so the individual digits form columns, add digits in one column at a time from right to left, and if there's an overflow, carry it to the next column to the left. Consider, for example, how 17 and 18 are added in the decimal numbering system:

```
   1
   17
 + 18
   ==
   35
```

First, 7 and 8 are added, producing 15. Since 15 is greater than 10, the 5 in the ones place is written down and the 1 in the tens place (shown shaded here) is carried over to the next column. Adding the two numbers in the leftmost column and the 1 carried from the previous column yields 3, giving us a final result of 35.

Now let's consider how a computer adds together the binary numbers 0111 and 0010:

```
   110
   0111
 + 0010
   ====
   1001
```

The procedure is exactly the same, but now a 1 is carried when two or more 1s are encountered. Since the binary value 0111 is equal to 7 decimal, 0010 is equal to 2, and 1001 is equal to 9, the computer just calculated that 7 plus 2 equals 9. Simple stuff, really. I remember learning base-2 math back in elementary school and wondering what on earth I was ever going to need it for. Now I know.

LOGIC GATES The previous section reveals in an abstract way how computers add binary numbers, but it says nothing about the circuits involved. The second step in understanding how a computer adds is learning about *logic gates* and *truth tables*. A logic gate is a simple electrical element that takes one or more inputs and generates an output whose value depends on the values of the inputs. A gate's truth table defines what output will result for each combination of inputs. Input and output values are represented by voltages. Typically, 5 Volts represents a 1 and 0 Volts represents a 0. We'll use 0s and 1s rather than voltages for our discussions because they more clearly illus-

14(9)

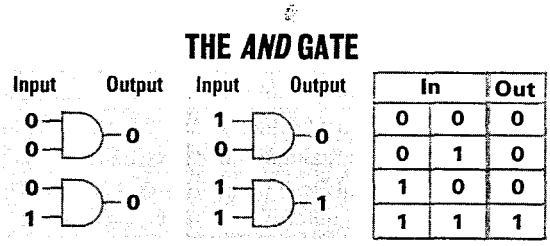


Figure 1: **The AND gate accepts two inputs, each with a value of either 1 or 0, and outputs 1 only if both inputs are equal to 1. The truth table lists the output value for each possible combination of input values.**

trate what the computer is doing.

Figure 1 illustrates the AND gate, which is one of the five basic gates used in digital electronics. The AND gate accepts two inputs, each a 1 or a 0, and outputs a 1 if (and only if) the first input *and* the second input are equal to 1. Any other combination of input values produces an

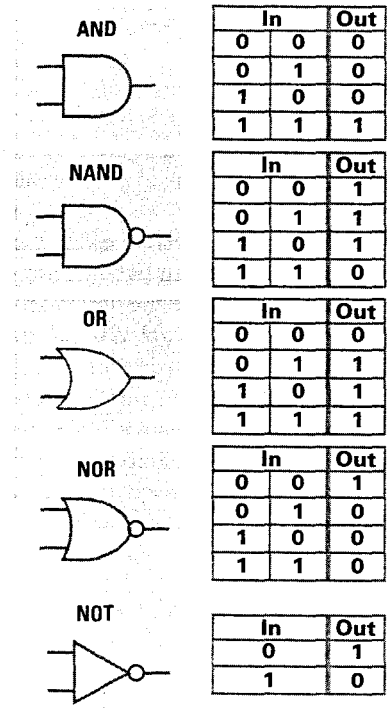


Figure 2: **Schematic representations of AND, NAND, OR, NOR, and NOT gates, and the corresponding truth tables.**

output value of 0. Four different combinations of 0s and 1s can be presented at the two inputs, so the truth table for an AND gate has four entries. If you know the values of the inputs, you can consult the truth table and determine the value of the output. Schematically, an AND gate is represented by a symbol that resembles an arched doorway turned on its side, as shown in Figure 1.

Figure 2 shows all five basic logic gates, the symbols used to represent them in logic diagrams, and their associated truth tables. An OR gate takes two inputs and outputs a 1 if either or both inputs is equal to 1. In other words, it outputs a 1 if input A *or* input B has a value of 1. NAND and NOR gates are the opposite of ANDs and ORs. Wherever the AND or OR gate produces a 1, the NAND or NOR produces a 0, and vice versa. A NAND gate outputs a 0 if and only if both inputs are equal to 1, while a NOR gate outputs a 0 if either or both inputs are 1. The NOT gate simply inverts its input, changing a 0 to a 1 or a 1 to a 0. Since AND and NAND gates produce exactly the opposite output given the same set of inputs, a NAND gate can be built by combining an AND gate and a NOT gate. (Sometimes the reverse is done: An AND gate is built from a NAND and a NOT.) Similarly, a NOR gate can be put together from an OR gate and a NOT gate, or vice versa.

The basic building block for all logic gates is the transistor, which, like a light switch, performs a simple on/off switching. All transistors have three terminals: a *gate* (not to be confused with *logic gate*), a *source*, and a *drain*. The transistor lets current flow between the source and the drain depending on the voltage supplied to its *gate*. There are several types of transistors, and many different ways to combine them to form logic gates.

Figure 3 shows a common way in which four transistors are combined to form a NAND gate. I've represented the type of transistor known as an *n*-channel MOSFET (metal-oxide semiconductor field-effect transistor) as a switch in a circle and an *p*-channel MOSFET in a square. An *N*-channel MOSFET *closes*—or allows current to pass through it—when a 1 is input to its gate, but *opens*—or impedes current flow—when a 0 is input. *P*-channel MOSFETs behave exactly the opposite, closing when a 0 is input and opening when a 1 is input. Figure 3 also illustrates how presenting 1s at both of the gate's inputs completes a circuit connecting the output line to the ground, producing a 0 at the output—exactly the behavior one would expect from a NAND gate. Note that there are no direct connections between the input and output lines. The input lines simply control the circuit's transistorized switches, which in turn control the current to the output lines.

You can verify that other inputs produce the proper results by diagramming the switch settings for various combinations of input values and following the resulting path backward from the output line to the source. In particular, notice that setting either input to 0 closes a square switch and sets the output to high, or 1. Setting both inputs to 0 closes both square switches, once again producing a 1 at the output.

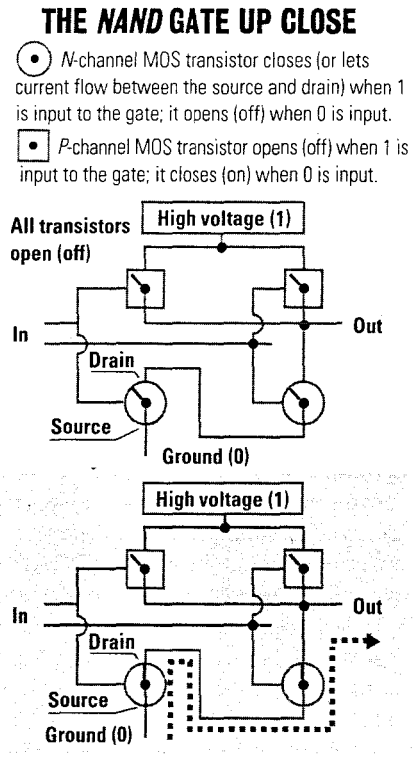
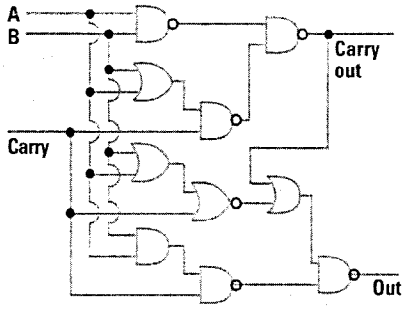


Figure 3: **This arrangement of transistors produces a NAND gate. The bottom diagram illustrates how setting both inputs to 1 forms a path from the low voltage source to the output line, producing a 0 at the output.**

FROM LOGIC GATES TO ADDERS So far you have learned that computers add numbers by adding individual 1s and 0s, and that logic gates allow us to convert



A	B	Carry	Out	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 4: **This simple 1-bit adder combines logic gates to produce a circuit that adds two binary digits and a carry value generated from a previous addition.**

various combinations of 1s and 0s into outputs equaling 1 or 0. If you suspect that somehow these logic gates are combined to produce circuits that add digits representing binary numbers, you're on the right track.

Figure 4 shows how one AND gate, five NAND gates, three OR gates, and one NOR gate can combine to form a circuit known as an *adder*. The circuit takes three inputs, which are labeled A, B, and CARRY. A and B are the two digits being added together; CARRY is the value carried from the previous column (0 if no carry). The adder has two outputs: OUT and CARRY OUT. OUT is the output digit, and CARRY OUT indicates whether a carry occurred as a result of adding A, B, and CARRY. Figure 4 also shows the adder's truth table. Setting all three inputs to 0 produces output and carry values equal to 0. Similarly, setting any one of the inputs to 1 produces a 1 output and a 0 carry; setting two inputs to 1 produces a 0 output and a 1 carry, and setting all three inputs to 1 produces a 1 output and a 1 carry. If you think about it for a moment, you will realize that this exactly matches the results you get when you add binary digits.

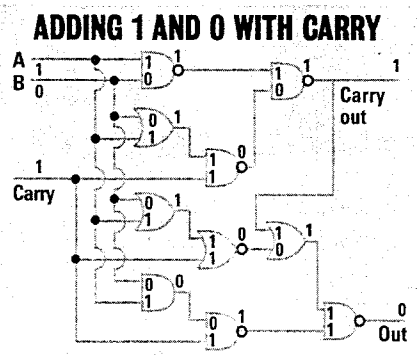


Figure 5: **The intermediate and output values produced when a 1 bit is added to a 0 bit with a carry bit equal to 1.**

Don't take the truth table in Figure 4 on faith. If you wish, you can verify its accuracy by sketching the logic elements on a piece of paper and computing what the outputs are for various input values. Figure 5 shows the inputs and outputs of each gate when A and CARRY are 1 and B is 0. Knowing the values input to a particular gate, you can use the truth tables in Figure 2 to determine the gate's output. Propagate these outputs throughout the circuit from beginning to end and you will eventually derive the output values for OUT and CARRY OUT. Do it for all eight possible combinations of input values and you'll generate the truth table entirely on your own.

It's but a small step from the 1-bit adder shown in Figure 4 to a 4-bit adder capable of adding larger numbers. A 4-bit adder is merely four 1-bit adders connected together in series, with the CARRY OUT line from the previous adder connected to CARRY on the next. Figure 6 shows what such an arrangement looks like schematically. In order to avoid clutter, each 1-bit adder is now shown as a block with labeled inputs and outputs. Figure 6 also shows the values of the inputs, outputs, and carries when 0111 is added to 0010. The result—1001, which you can see on the four output lines—matches the result obtained earlier when we added the same two numbers by hand.

Engineers can build adders capable of handling any number of digits by stringing together enough 1-bit adders. That's basically how it's done in modern microprocessors, although their adding circuits are much more sophisticated than this. Among other things, they incorporate extra logic permitting several bits to be added simultaneously, which greatly reduces the computation time. Circuits perform subtraction in much the same way as addition, with carry lines becoming borrow lines signifying a "borrow" from the next column.

It's all pretty simple when you view it at this level. When it becomes daunting-

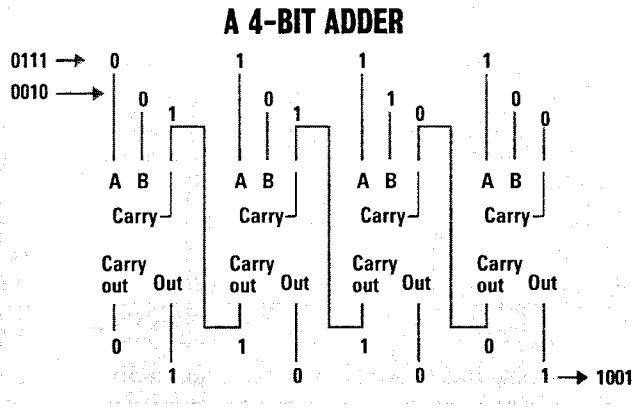


Figure 6: **Four 1-bit adders connected in series to form a 4-bit adder. The outputs shown here are the result of adding 0111 and 0010.**

ly complex is when you think of the millions upon millions of transistors present in a typical microprocessor chip. It's no wonder the world recently learned that the Pentium has a division bug in its floating-point unit. The real wonder is that human error has not introduced major design flaws in microprocessors before now. Or maybe it has, and we just haven't discovered them yet.

But that, dear reader, is a topic for another day.

FURTHER READING If you want to delve deeper into the topic of transistor logic, try the following books: *How Computers Really Work*, by Milind S. Pandit (Osborne McGraw-Hill, 1993; ISBN: 0-07-881936-9); and *Understanding Solid State Electronics*, Fifth Edition, by Don L. Cannon (SAMS Publishing, 1991; ISBN: 0-672-27338-1). □