THE DRAWING BOARD

Working with counters

ROBERT GROSSBLATT

IC-FABRICATION TECHNOLOGY HAS COME a long way since the first IC rolled off the production line a mere twenty or so years ago. Component density has gone from four transistors on the early chips to over four hundred thousand transistors on current ones. These mind boggling numbers have led to all sorts of good things—from five-dollar microprocessors to blister-packed digital watches sold next to the canned soup in the supermarket. The result of all this on someone (like me) who occasionally likes to re-invent the wheel to solve circuit problems has been quite extraordinary.

I've had to re-define the wheel.

What is new, expensive, and exotic today is most definitely cheap and ho-hum tomorrow. I can remember using loads of power-gobbling gates and flip-flops to build counters. Today that approach to a circuit design would be ridiculous because the array of features in available MSI (*M*edium *S*cale *I*ntegration) counters can take care of any design problem you can imagine. Counters have to be considered a basic building block of digital design—in other words, a one IC addition to a circuit.

Now, the word "counter" takes in a lot of territory—anything that does first one thing and then another in a pre-arranged sequence can be called a counter. Just about the only thing they have in common is that they need a power supply and some sort of clock. There are lots of ways you could divide them up but since we're calling them a basic building block, we'll make a basic two divisions—counters with a one-and-only-one type of output and those with encoded outputs.

Every logic family has its own array of counters and for our purposes, anything we say about the counters in one family will be more or less true of the counters in any other family. We'll restrict our discussion to CMOS counters since we're more interested in finding out how to use them than in chopping the top off the package and looking at the silicon.

The 4017 is a good example of a counter that has only one output decoded at a time. It has ten outputs and they go high one at a time in fixed sequence as long as the ENABLE and RESET pins are held at ground. A high on the ENABLE pin will disable the clock input and the counter will ignore incoming clock pulses. A high on the RESET pin will make the "0" output go high; it will stay that way until the RESET pin is grounded again. There's also a "carry" output that divides the input clock by ten—it's high for counts zero through four and low for counts five through nine. This IC is really a shift register with a few added bells and whistles. There are, however, some interesting things we can learn from it and some extremely useful things that it can do when we put it to work for us

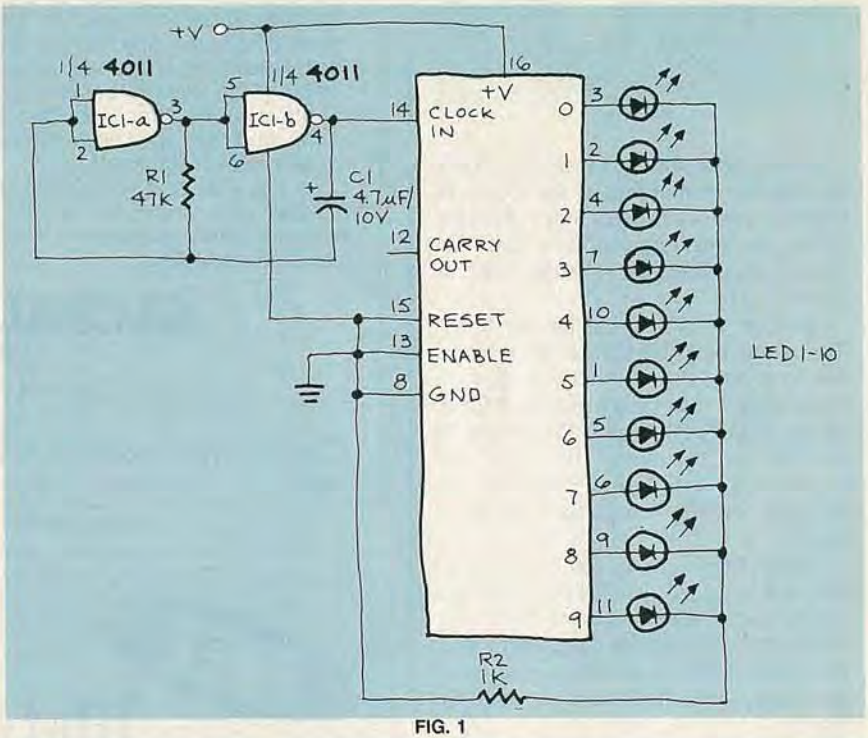

FIG. 1

First of all, this is a synchronous counter. That means that all the internal flip-flops are triggered by the incoming clock at the same time. The other possible arrangement is called a ripple counter, meaning that the internal clocking takes place like a row of dominoes—each stage triggers the next stage. Ripple counters are cheaper to make; but they're much slower than synchronous ones since stage changes happen in serial, rather than parallel, fashion. They will also temporarily output incorrect counts while the dominoes are still falling. That glitchy period is euphemistically called the "settling time" but it would be more accurate to call it the time when the output of the counter was just plain wrong. Since the speed at which CMOS operates is a function of, among other things, the supply voltage, lower voltages can lead to delays many microseconds long. During those microseconds the counter output is not exactly something you'd want to take to the bank.

The one-and-only-one type of counter can come in really handy when you have to solve certain design problems. The keyboard data encoder we designed showed two of the many possible uses for this type of counter. We used it there to select a particular switch at the keyboard and also as a sequencer to control the order in which data was latched onto the

TABLE 1

Operation	Propagation Delay	Pulse Width	Transition Time
Decoded output	500 nanoseconds	200 nanoseconds	300 nanoseconds
Reset	450 nanoseconds	200 nanoseconds	250 nanoseconds

bus. That is, of course, by no means all it's good for.

The best way to understand how the IC is used is, naturally enough, to actually use it. Since the 4017 has outputs that sequence one after another, probably the most basic circuit we can build is the sequencer shown in Fig. 1. We're using one half of a 4011 to make a simple clock we can use to drive the 4017. Any other oscillator would be just as good. The frequency of the 4011 clock follows the form $F = 1/1.4RC$. Since we want to be able to see the 4017 outputs in action, we'll pick values for the clock components that slow it down enough for us to watch things happen. The values shown will give a clock frequency of about 3 Hz—a nice compromise between visibility and impatience.

Everything else in the circuit is straightforward. By tying both the ENABLE and RESET pins to ground, the 4017 will count from zero to nine over and over again. Now, that isn't the most exciting thing I've ever seen but even this circuit has some important real-world uses. What you're looking at is a one-IC method of delaying clock pulses by a time period exactly equal to N clock pulses. All you have to do is route your clock to the input of the 4017 and pick off whichever phase-shifted output you want. Of course your input clock will have to be running ten times faster than the frequency you want to see at the output, but that's not much of a problem.

We can spice things up even more by using the ENABLE and RESET pins. Tying the ENABLE pin to a particular output means that the 4017 will count to a certain number and then stop. Doing the same thing with the RESET pin will give you a really down-and-dirty method of frequency division. Since the IC will reset to zero whenever the selected output goes high, any of the chip's outputs in sequence before the selected one will go high at a rate equal to f/N where f is the input clock frequency and N is the number you're dividing by.

Someone once said that there's no such thing as a free lunch and that applies here as well as anywhere. While it's obviously true that you can divide a clock down this way, it's also unfortunately true that you're paying a price for simplicity. First, the duty cycle of the output will be something like 1/N. This makes sense because the outputs go high for one full cycle of the input clock and remain low for the rest of the time. I said "something like 1/N" because there's a certain amount of uncertainty that's caused by the weirdness that goes on when the selected output goes high and the IC resets. That leads to the second price we have to pay.

When you operate the IC at 5 volts, the propagation delay (the time it takes for the IC to change to a new state) from one output to the next is about 500 nanoseconds. This means that there will be a 500-nanosecond delay between the time an input pulse is detected and the IC puts a high on the next output in sequence. Let's assume we have the RESET pin tied to output 4 and output 3 is high. Along comes the next input clock pulse—it's detected and the internal machinery of the IC starts to decode it. When that operation is finished it simultaneously turns off output 3 and turns on output 4 (remember that this is a synchronous counter). So far so good.

When output 4 goes high, it brings the RESET pin high and causes the IC to turn off output 4 and put a high on pin 3—the first output in its sequence. The problem crops up because the 4017 features asynchronous reset. That means that reset takes place whenever the RESET pin is brought high. In an IC with synchronous reset, the reset operation wouldn't happen until the next clock pulse arrived at the input. The 4017 is counting as a synchronous counter but reset is happening in a ripple fashion. Our problem is that the IC ignores incoming clock pulses when the RESET pin is high as well as during the entire reset operation. A quick look at Table 1—which shows us the characteristic operational times for a 4017 operating from 5 volts—illustrates exactly what the problem is.

In the best of all possible worlds, therefore, there's a built-in period of almost 1 microsecond (500 + 450 nanoseconds) during which the 4017 is performing its reset operation. We have to wait for the selected output to be decoded and then twiddle our thumbs while the reset operation is carried out. Since the clock input is disabled half this entire time (during reset), we'd better make sure that no clock pulses show up at the input because they're going to be ignored. The price, therefore, that we're paying for down-and-dirty frequency division is a cutback in the maximum input frequency we can have and the possibility of glitches in the count.

Next month we'll see how to add synchronous reset and take care of these other problems by a little creative gating. We'll also start designing a circuit that will not only divide frequencies by any number we want, but is keyboard-programmable as well. **R-E**