what is a ROM?

ROM's have a fantastic number of uses and are widely available as you-build-it and factory-builds-it types. Here's what ROM's are and what they are good for.

How would you like to build your own integrated circuit, perhaps to do a job you can't find some catalog item for? This used to cost $15,000 or so and take months of work. Today you can do it for $5 in minutes, with surplus units, and under $20 with first-run parts. The trick is to use an extremely versatile integrated circuit called a Read Only Memory or ROM for short. Let's take a closer look at this exciting integrated circuit and see what it is and how you can use it.

Actually, it would be much better if a ROM were called something else, for its name implies it's only good for computers. Worse yet, its name says there is something "wrong" or incomplete with the device. It would be best to call a ROM a "universal code, state, logic, or sequence converter", for this name at least hints at the thousands of different things you can do with the same basic device, custom-programmed to do a specific job. Since ROM is easier to say than "UCSLOSC", we'll go along with the original name.

Figure 1 shows the important parts of a ROM. There are a number of *input* lines, a series of *output* lines, some power connections, and

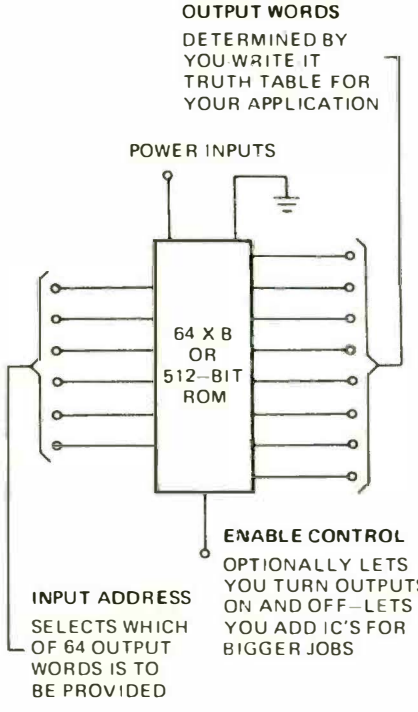
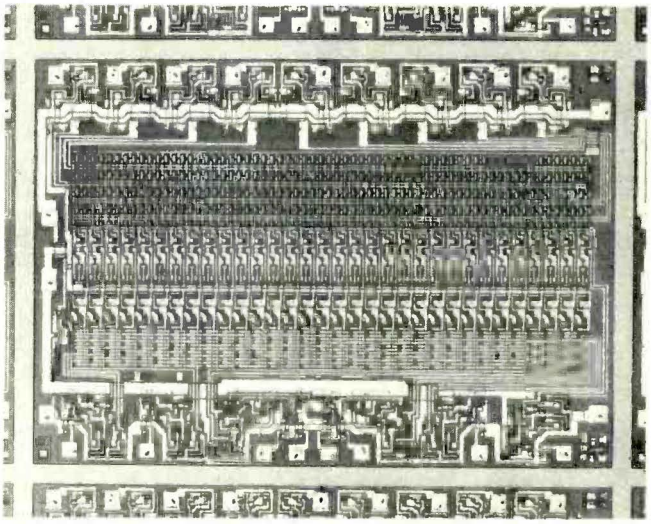


FIG. 1—ESSENTIAL PARTS of a typical ROM. This example is medium-sized and stores 512 bits of custom-programmed decisions by providing 64 possible 8-bit words at the outputs.

an *enable* control that optionally lets you turn the outputs on or off or combine them with other ROM outputs in other packages. As you've probably suspected, a ROM is a *digital* device, meaning it accepts yesses and no's or 1's and 0's or positive voltage and ground as two-state input signals. It provides similar 1s and 0's as two-state output levels. ROM's are available in most every logic family, including TTL, PMOS, CMOS, and ECL.

For each and every unique combination of input ones and zeros, a code word appears on the outputs. What this code word is or what it does in the rest of the circuit is *yours* to decide, for you can *teach* a ROM to do any one specific job for you.

For instance, if a ROM has six input lines, there are 64 (2^6) possible combinations of ones and zeros on the inputs, ranging from 000000,



INSIDE AN ROM you'll find a labyrinth of individual memory cells. Remember, actual size of this assembly is about .02 inch wide.

000001, 000010, through 111110 and 111111. For each of these possible 64 conditions, you can select *any* output word you like, its maximum length determined by the number of available output leads. If you have eight output leads, each of the 64 words you select can be up to 8 bits long. Since we have 64 possible 8-bit words, we apparently have an internal ROM "decision" or "training" capability of 512 (8×64) *bits*. Each of the 512 locations can be a one or a zero per your choice, so there are apparently 2^{512} or *billions upon billions* of *different* things you can teach one IC to do.

The arrangement of the ones and zeros you want is usually shown on a *truth table*, a state-by-state listing of all possible input combinations and the desired outputs you want.

How is this teaching done? There are several basic ways. If you need a lot of identical ROM's and are sure of what you want, you use a *mask-programmable* ROM. Here a final metal overlay connection pattern is set up for your particular program. All the ROM's made are identical up to this step. Your mask then customizes your order to the particular truth table you need.

More popular is the *field-programmable* ROM. You use these if you only need one or two, or aren't sure if your truth table will work, or suspect you will have to change things later. Some field-programmable ROM's arrive from the factory with a fuse at *each* possible location in the memory. Before you use the ROM, you go through a *programming* procedure that selectively blows out the fuses you don't want, leaving you with a custom pattern of ones and zeros that matches your truth table. You do the programming one bit at a time, usually applying a current of several hundred mA at a programming input. The current is increased till the fuse opens, and you then go on to the next fuse you want to open.

All this really takes is a variable power supply with a meter, but the "zero defects" nature of this work and its "up the wall" aspects make programming services very desirable.

Many electronic distributors offer nominal or free programming services and guarantee the results—provided, of course, that you wrote the truth table down correctly! Programming machines are also available that ease the problem. These cost several hundred to several thousand dollars, but speed up the programming tremendously and eliminate many error possibilities.

Other field-programmable ROM's use buried charge (electret style)

layers or silicon bridges instead of nichrome fuses, but the result is the same. The ROM fresh from the factory is either all ones or all zeros, and you do something—usually by applying an excessive voltage or current to remove or implant something at every memory location—that changes the ones to zeros or vice versa. You then end up with the truth table you want.

Once programmed, the majority of ROM's are *permanent*; hence the name *read only*. If you made a mistake, you throw the IC away and start on a new one. On the other hand, since the programming is mechanical, it's forever independent of supply power. Turn your ROM off for a year and reapply power—and the truth table is still inside. A few newer ROM's are erasable by removing part of the lid and applying intense ultraviolet light. These are expensive and not too common yet.

Building your own read only memory

Let's build a "semi-discrete" ROM and see what it can show us about how the real ones work. Outside of doing it once as an exercise or to learn more about the process, going this route is complex and expensive compared to using the real thing.

Suppose we need a way of converting a 4-bit *hexadecimal* number into a 7-segment display so we can display the numbers 0, 1, 2, 3 . . . 9, A, B, C, D, E, and finally F with the letters handling states 10 through 15 and the numbers representing their own binary equivalents. A quick check of catalogs will turn up lots of different decoder/driver integrated circuits. This particular one seems to be rare, so let's pretend it doesn't exist at all. We have to use a ROM to build it.

Note that we'd go up the wall trying to build this out of simple gate packages—it would take a bunch of them and the design would take hours. With a ROM, the design only takes minutes, and a one-package solution almost always results.

We start by generating a *truth table* (Fig. 2). Our four input lines have 16 possible states (0000, 0001 . . . through 1111). We need

INPUT				OUTPUT								PATTERN
D	C	B	A	A	B	C	D	E	F	G	H	
0	0	0	0	1	1	1	1	1	1	0	0	
0	0	0	1	0	1	1	0	0	0	0	1	
0	0	1	0	1	1	0	1	1	0	1	1	
0	0	1	1	1	1	1	1	0	0	1	1	
0	1	0	0	0	1	1	0	0	1	1	1	
0	1	0	1	1	0	1	1	0	1	1	1	
0	1	1	0	1	0	1	1	1	1	1	1	
0	1	1	1	1	1	1	0	0	0	0	1	
1	0	0	0	1	1	1	1	1	1	1	1	
1	0	0	1	1	1	1	1	0	1	1	1	
1	0	1	0	1	1	1	1	1	0	1	1	
1	0	1	1	0	0	1	1	1	1	1	1	
1	1	0	0	0	0	0	1	1	0	1	1	
1	1	0	1	0	1	1	1	1	0	1	1	
1	1	1	0	1	1	0	1	1	1	1	1	
1	1	1	1	0	0	0	1	1	1	1	1	

A
F G B
E C
D

1 = LIT SEGMENT

FIG. 2—THE TRUTH TABLE WE NEED to build a 7-segment decoder/driver with a ROM. Its four input lines have 16 possible states. Each of its seven output lines drives one segment of a 7-segment display device.

seven output lines—one for each segment of the display. Let's provide eight to round things out and put a "count zero" detector on the eighth line. Each output line lights a display segment if it is positive and puts out a display segment if it is grounded.

For instance, we could connect our output lines to a MAN3 or MAN4 common-cathode LED readout. Positive current lights a segment. Voltage near ground puts it out. The segments are labeled A through G in the usual clockwise from the top manner.

To build the actual ROM, we use a bunch of diodes and a 74154 4-line-to-16-line decoder. This particular IC converts the four input lines into a one-down-out-of-sixteen pattern on our *intermediate output* lines, so that a 0000 input grounds the top intermediate output, a 0001 the next one down, and so on down to 1111 which grounds the bottom intermediate output line. Only one output line is grounded at a time; the rest remain positive. Figure 3 shows the circuitry.

Going to our truth table, 0000 should give us an output 0, lighting

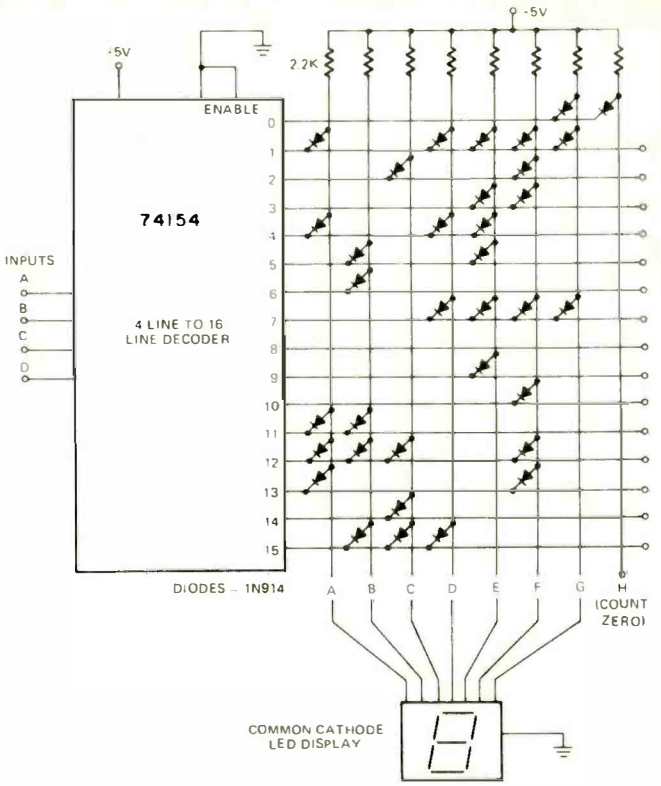


FIG. 3—OUR "FAKE" ROM is made from a 74154 4-line to 16-line decoder and a diode matrix to turn on and off the segments of a cold-cathode or LED 7-segment display device.

every segment but G, so we put a diode between line G and the 0000 decoded output. This diode conducts only on count 0000 and puts out this segment only at that time. On 0001, we only want to light B and C, so we evidently have to put zeros and diodes on A, D, E, F, and G. On 0010, to get a 2 lit, we put diodes on C and F. And so on, down the truth table.

We mathematically generate the truth table by placing a 0 everywhere we want a segment out and a 1 everywhere we want a segment lit. We physically program our ROM by putting a diode everywhere we want a 0 and leaving a diode off everywhere we want a 1. And this completes our decoder/driver.

To get fancy, we can use the eighth output lines as a state 0 decoder that might be useful for blanking or somewhere else in the system. All this takes is a new diode on the 0 line. We can use the output enable on the 74154 to drive all the outputs high for a lamp test, and we can blank the display either by breaking ground or removing the supply power.

We used diodes to teach our ROM to do one specific thing. There are 128 possible memory locations in our simple ROM. Each of these locations can be given a 1 (no diode) or a 0 (diode) per your choice, so there are apparently 2^{128} *different* truth tables you can write. (My math book stops at 2^{101} = 2,535,301,200,456,458,802, 934,406,410,752. 2^{128} could be 134,317,728 *times* as large as this—you figure it out.)

Obviously we have a bunch of different truth tables we can write—A great heaping bunch. We can teach the ROM anything we like, consistent with the available number of inputs and outputs.

For instance, we could tell the ROM to subtract 3 from each input. Or multiply by 6.2. Or take the square root of it. Or we could tell it to decode and combine only certain states. We could make it play music. We can change codes or number systems. We can generate waveforms. There doesn't have to be any clear cut rhyme or reason relationship between inputs and outputs. If you can draw a truth table, the ROM will do the job for you—quickly and in a single package.

The *organization* of this particular ROM is called 16×8 or 16-8-bit words. Its potential memory locations are 128, so it is also called a 128 *bit* ROM.

ROM design is philosophically very different than older logic designs. The name of the game used to be a thing called "minimization", where you tried to get the logic equations in their simplest form and then build up a pile of gates to realize the "simplest" possible form. With ROM's you use *redundancy* instead. You take one logic

block in one integrated circuit package with an incredible amount of redundancy—it can realize the "minimum" equivalent of *any* and *all* possible equations you could care to write consistent with the available inputs and outputs. You *ignore* the math and the simplifications! Instead, you just write down the truth table you want and program the ROM.

The benefits of redundant circuit design are overwhelming. The old way, you got a "minimum" logic design that took a dozen packages and took hours to design and debug. It was essentially unchangeable after design, particularly once it was locked into a PC board. The new way takes only seconds. Write your program, program the ROM and plug it in. The new way always works without any worry about glitches, races, disallowed conditions, sub-routines, and similar horrors. Changes? Simple. Just take out the old single IC that does the job and put a new one in its place. For everyday logic use, the textbook "minimization" techniques are an inexcusible waste of time and money once you get past a two- or three-package gate complexity. All they "minimize" is profits and the probability of success.

Commercial availability

Table I lists the commercial sources of programmable ROM's, one source of programmers, and one distributor that does programming. Table 2 lists a number of common ROM's and their organizations.

TABLE I

Some sources of ROM's and services:

CIRCUITS

Harris Semiconductor
Box 883
Melbourne, Florida, 32901

Motorola Semiconductor Products
Box 20912
Phoenix, Arizona, 85036

Intel Corp.
3065 Bowers Avenue
Santa Clara, California, 95051

National Semiconductor Corp.
2900 Semiconductor Drive
Santa Clara, California, 95051

Intersil, Inc.
10900 N. Tantau Avenue
Cupertino, California, 95014

Signetics
811 East Arques Avenue
Sunnyvale, California, 94086

Microsystems International
Box 3529 Station C
Ottawa, Canada

Solitron Devices
8808 Balboa Avenue
San Diego, California, 94086

Monolithic Memories, Inc.
1165 East Arques Avenue
Sunnyvale, California, 94086

Texas Instruments Inc.
Box 1443, Station 612
Houston, Texas, 77001

PROGRAMMING MACHINES

Spectrum Dynamics
2300 East Oakland Park Blvd.
Ft. Lauderdale, Florida

PROGRAMMING SERVICES

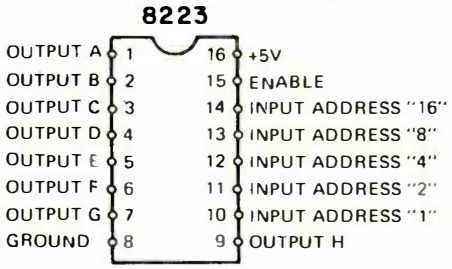
Semiconductor Specialists
Box 66125 OHare Airport
Chicago, Illinois, 60666

TABLE II

Here are a few currently popular programmable ROM's:

Part Number	Manufacturer	Organization	Bits
HROM-1-0512	Harris	64×8	512
HROM-1-1256	"	256×1	256
HROM-1-8256	"	32×8	256
HROM-1-1024	"	256×4	1024
HROM-1-2048	"	512×4	2048
IM5610	Intersil	32×8	256
IM5623	"	256×4	1024
MCM5003	Motorola	64×8	512
MCM5005	"	256×4	1024
MCM10139	"	32×8	256
MCM10149	"	256×4	1024
N8223	Signetics	32×8	256
N82S26	"	256×4	1024
SN74186	Texas Insts.	64×8	512
SN74188	"	32×8	256

One programmable ROM that's showing up quite a bit in the surplus market recently is the Signetics 8223. Costs have gone as low as $5 each. It is shown in Fig. 4. It's a bipolar device, DTL and TTL compatible and works with a single +5-volt supply. Operating speed is a fraction of a microsecond.



TOP VIEW

FIG. 4—A POPULAR PROGRAMMABLE ROM that is inexpensive, readily available, bipolar, and TTL compatible. Supply voltage is +8 V supply current is 65 mA. Connecting "enable" to ground provides an output code. Connection to +volts floats the output.

By the way—*preprogrammed* ROM's available surplus are only useful if you know *exactly* what they are and what they can do for you—a random or unknown program is totally worthless and essentially impossible to decode.

When do you use a ROM?

You use a ROM anytime you want a group of input numbers to be somehow related to a second group of output numbers, especially when you can't find a stock IC to do the job. ROM's become particularly attractive if the job seems hopelessly complex for construction using gate packages.

There are several different ways to use your input numbers. If you feed your ROM one number and then get a new one out on a random basis, you are using the system for *code conversion* or *table lookup*. If you sequentially go through your inputs, you have a *waveform generator*. Here the outputs provide an orderly progression of state changes, perhaps to generate a sinewave or a music note. If you use your inputs as separate logic inputs instead of feeding them a whole word at a time, you have a *programmable logic array*. Similarly, if you route your outputs to separate and distinct places, you have a *sequencer*, a *controller*, a timing generator, or a rhythm generator.

To really get fancy, you can let a ROM control *itself*. To do this, you store or latch the outputs each cycle and use the *last* output to provide the *next input* address. This way, the ROM marches through its truth table in a prescribed and controlled way. You use this for unusual length *counters* and computer *microprogramming*.

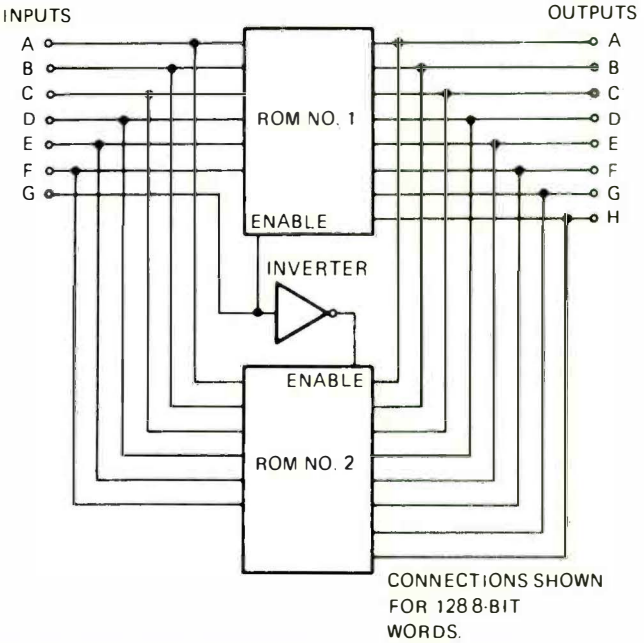


FIG. 5—HOW TO EXPAND ROM's by using several of them. Note that the doubled storage only offers one new input lead.

ROM design is easy. First, you make sure you really need one and that nothing is available commercially to do the job. Then you write your truth table. Then you find a ROM that fits it. Then you program the ROM.

If your truth table seems hopelessly large, you try to *minimize* it through several tricks of the trade. These include removing mirror images (such as generating only one quadrant of a sinewave), putting easy-to-realize functions *outside* the main ROM, using multiple trips through the ROM, factoring, rearranging to fit ROM organizations, eliminating "don't care" states, and so on. Virtually *every* truth table can be minimized in a complex system. If you have reduced things as far as possible and you still can't fit it in, you go to a larger ROM or several ROM's combined with input steering and output enables.

Note that you don't double the inputs when you add a second ROM to a first one—all you gain is one extra input. Since you only doubled the memory capacity, your addressing has only increased by one power of two. Seven lines have twice the storage capability of six. If you are using 6-input (64-word) ROM's, it takes *two* of them for seven inputs (128 words), *four* of them for eight inputs (256 words), *eight* of them for nine inputs, (512 words), and so on. Figure 5 shows how you combine ROM's with their enables.

There are several stock organizations of ROM's 16×8, 256×1, 64×8, and 128×4 being common smaller ones. Sometimes you can rearrange things with a latch or a data selector to change the organization if you want to. For instance, if your particular ROM has eight output leads, and you only need a 4-output word, you can use a 4-pole double-throw data selector (74157) to pick either the top or bottom four bits. This *doubles* the number of words you have available. On the other hand, you can provide two 8-bit latches on the output and enable them on *alternate* addresses. If you look at all 16 outputs at the right time, you get a 16-bit output word. Of course, to get this, you've cut the number of words in half and reduced the possible operating speed at the same time. Figure 6 shows how you can change the organization of a ROM to fit your needs.

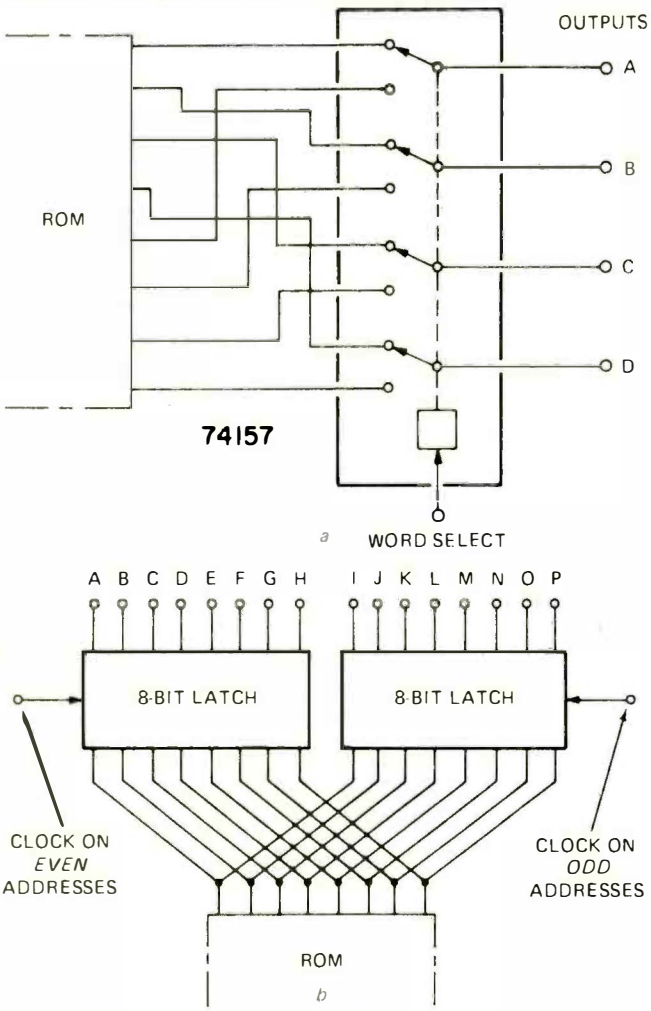


FIG. 6—THE DATA SELECTOR at (a) gives you twice as many output words of half the normal length. Using the setup at (b), the odd addresses are saved until even an address arrives. Output is half as many output words of twice the normal length.

Sometimes a special custom organization will help. This was done for the time-zone-converting ROM on the *Radio-Electronics* Superclock (July 1972), where a 384×6 MOS ROM was used with a simple external OR gate to convert 2400-hour time to any timezone in the world. "Non-binary" organizations normally cost quite a bit of money and are not available in field programmable units.

Even if you are planning on using a bunch of identical ROM's, you first use programmable ones, and then later go on to the cheaper mask-programmable versions. The breakeven point is typically several hundred identical units. A few dozen identical ROM's are easily copied or duplicated on a small programming machine. If you do go to mask-programmed units, ROM-PROM pairs make the changeover easy.

Stock read-only memories

Besides custom patterns, you can get stock pattern ROM's preprogrammed and ready to use. There is no masking charge for these, since they are a popular enough pattern that lots of them can be sold. Very common examples are the character generators such as we used in the *Radio-Electronics* TV Typewriter (September 73) and the TV Time Display (scheduled for a forthcoming issue).

There are several types of character generators. Most of them accept a 6-bit ASCII standard computer code on one set of inputs and some system timing on some remaining outputs. A *raw* output character generator is designed to work with TV sets. It puts out a bunch of dots or *undots* on its output lines. These go to a TTL shift register and are then clocked out as video. A *column* output character generator works sideways and puts out a vertical group of dots and undots useful for moving message signs and strip printers. Either type costs around $12, but a bunch of support circuitry is needed.

Other stock ROM's include code converters, particularly to get from the specialized SELECTRIC and FIODIC codes to ASCII and back again. Trig tables for sine and cosine generation are also fairly common, although still a bit steep in price. A vastly different stock ROM is the American Microsystems S2566 Rhythm Generator used to generate the accompaniment beats (waltz, tango, etc.) on an electronic organ.

In addition, many ordinary IC's are really ROM's in disguise, for the semiconductor people long ago found out that it's easier to design one redundant ROM pattern and then change the metallization overlay than to relay out and make separate IC's for each and every special function. The Motorola MC4000 series of TTL uses several ROM functions.

Applications

We've already seen that a ROM can be used anywhere you want to convert one group of digital words to a second group of words, either on a one-at-a-time, a sequential, or a let-the-first-one-decide-the-next-one basis. The wilder or the more unusual the relationship between the input and output, the better a ROM will work, for you work directly with the truth table. Competing systems require deriving all the individual logic relationships between input and output, and cannot normally be done in a single IC.

So far, we've talked about display decoder drivers, character generators, sinewave generators, electronic music, time-zone converters, and code converters. Let's take a quick look at some other possibilities.

Frequency synthesizers and digital programmers often use thumbwheel switches. The numbers of the switches indicate a channel number or a frequency, but the circuitry inside may take entirely different numbers to operate. Rather than use an expensive special switch, a ROM performs the internal conversion—the operator sees his number and the circuitry sees the number it needs at the same time.

Sinewaves are easy to generate by taking a counter and a sine-lookup ROM. Add a D/A converter for a low-distortion sinewave oscillator of constant amplitude that can go down to ultra-low frequencies without any large parts.

ROM's are used in cathode ray tube display systems for pincushion correction, dynamic focus and convergence, and so on. Besides generating dot-matrix characters, ROM's can store and generate whole messages as well. Often you generate the fixed portions of a message in a ROM and add the changing part to it. You can also scramble or unscramble data with ROM's, throwing away what you don't want and rearranging things to get a needed format.

Any logic equation you can write in truth-table form is also easily handled by ROM's. The one-package solution and instant design are top advantages. Besides, the circuit is trivially easy to change—you simply replace the ROM. Compare this with a traditional "minimum" logic design of several dozen packages and locked-in interconnections.